Google App Engine

Construindo serviços na nuvem

@tenebroso

PAULO SIÉCOLA

SOBRE O LIVRO

O **Google App Engine** é uma plataforma de computação nas nuvens que permite a execução de aplicações Web na **infraestrutura do Google**. Tudo isso de forma fácil e escalável, sem a necessidade de manutenção em sistemas operacionais ou servidores.

Ele possui várias opções de utilização **gratuitas**, baseadas em cotas e limites, que permitem o desenvolvimento de pequenas aplicações para testes e estudos sem gastar nenhum centavo! Isso torna a plataforma muito atrativa se você está começando e deseja aprender mais sobre ela.

Este livro aborda vários aspectos dessa plataforma, principalmente aqueles relativos ao desenvolvimento de aplicações em Java para interagir com seus recursos, e também a administração deles através das ferramentas disponibilizadas pelo Google App Engine, ou GAE, como é comumente chamado.

Os principais tópicos deste livro são:

- Conceitos básicos do Google App Engine;
- Como desenvolver e hospedar **serviços REST** com Jersey no GAE;
- Como trabalhar com o Google **Datastore**;
    - Entidades
    - Consultas
    - Administração
- Como desenvolver uma aplicação no GAE utilizando o **Google Cloud Messaging** para enviar mensagens a aplicativos móveis;
- Como **agendar tarefas** no GAE para invocar um

serviço em sua aplicação;

- Como utilizar OAuth, como mecanismo de autenticação de usuários, para proteger o acesso aos serviços da sua aplicação;
- Como gerar, visualizar e gerenciar os logs das aplicações Java hospedadas no GAE;
- Como trabalhar com *memory cache* para armazenar dados temporários em memória de forma rápida, mas não persistente;
- Como visualizar e gerenciar os erros gerados pela aplicação.

Ao longo do livro, alguns projetos Java serão criados para o GAE. O código-fonte deles estão no GitHub, no seguinte endereço: https://github.com/siecola/GAEBook.

Para o desenvolvimento desses projetos, você poderá usar a IDE **Eclipse**, juntamente com algumas ferramentas fornecidas pelo Google. Há um capítulo dedicado à preparação do ambiente de desenvolvimento.

Você também pode participar do grupo de discussão deste livro, deixando comentários, dúvidas ou sugestões. O link é: http://forum.casadocodigo.com.br.

A quem se destina este livro

Esse livro é útil para desenvolvedores de aplicações Web que desejam conhecer sobre a plataforma de computação nas nuvens Google App Engine. Será possível aprender a trabalhar com suas tecnologias, ferramentas e técnicas para construir sistemas arquitetados para serem escaláveis. Para administradores de sistema, este livro traz tópicos essenciais para aqueles que desejam administrar aplicações que serão hospedadas no Google App

Engine, pois há uma boa parte do conteúdo dedicado a isso.

É interessante que o leitor possua familiaridade com **Java e programação orientada a objetos**, bem como com a IDE Eclipse, para poder aproveitar com mais intensidade o material apresentado, e se aventurar nos exercícios propostos. Porém, o livro aborda todos os conteúdos de forma didática, construindo os **exemplos desde o início** e detalhando os conceitos a partir de um nível que possa ser compreendido por programadores com qualquer nível de experiência.

Sobre o autor

Paulo César Siécola é Mestre em Ciência da Computação pelo Instituto de Matemática e Estatística da Universidade de São Paulo (2011). Possui graduação em Engenharia Elétrica pelo Instituto Nacional de Telecomunicações - INATEL (2005).

Atualmente, é Especialista em Sistemas Sênior no Inatel Competence Center e Professor em cursos de Pós-Graduação no INATEL. Tem experiência em desenvolvimento de software em **C, Java e C#**, atuando principalmente nos seguintes temas: desenvolvimento Web, sistemas embarcados, análise de protocolos de redes de computadores e desenvolvimento de aplicações para GNU/Linux embarcado.

Agradecimentos

Gostaria de agradecer ao Adriano Almeida, pela oportunidade de publicar um livro na Casa do Código, e também agradecer a Vivian Matsui, pelo empenho e dedicação nas revisões didáticas.

Agradeço aos meus professores e mestres, pois, sem eles, não poderia compartilhar o conhecimento.

Agradeço meus pais, que são a fonte da minha motivação para o trabalho.

Obrigado meu Deus, pelos dons que de Ti recebi. Espero estar utilizando-os com sabedoria e equilíbrio.

Finalmente, agradeço a minha amada esposa Isabel, pela paciência, apoio e incentivo sem igual.

Sumário

Versão: 19.3.26

Capítulo 1

O QUE É GOOGLE APP ENGINE

O Google App Engine é uma plataforma de computação nas nuvens que permite a execução de aplicações Web na infraestrutura do Google, de forma fácil e escalável, sem a necessidade de manutenção em sistemas operacionais e servidores. Todas as aplicações ficam hospedadas sob o domínio appspot.com, e podem ser acessadas por todo o mundo ou apenas por pessoas/aplicações autorizadas.

As aplicações podem ser desenvolvidas em Java, Python, PHP e Go, com um ambiente de *runtime* específico para cada linguagem.

O estilo de cobrança do Google App Engine é o *pay as you go*, ou seja, você só paga pelo tanto que a aplicação usar de recursos da plataforma. Porém, é possível iniciar o desenvolvimento sem pagar nada, com limites razoáveis de banda, armazenamento e número de acessos por mês. Esses limites são suficientes para provas de conceitos, estudos e até mesmo aplicações de pequeno porte.

Durante este livro, será utilizada a linguagem de programação Java para o desenvolvimento de aplicações de exemplo, que serão hospedadas no Google App Engine. Por isso, o foco das descrições a partir desse ponto será exclusivo para essa linguagem.

1.1 SANDBOX

As aplicações no Google App Engine rodam em um ambiente seguro e controlado, que limita e controla o acesso ao sistema operacional onde está sendo executado. Tais limitações permitem que a infraestrutura controle melhor a alocação de recursos e máquinas para as diversas aplicações. Alguns exemplos dessas limitações são:

- As aplicações só podem ser acessadas por meio de requisições HTTP ou HTTPS em portas padrões;
- Não é possível escrever em arquivos. Para leitura, só é possível ler arquivos que foram carregados pela própria aplicação. Para armazenamento de dados, a aplicação deve usar os mecanismos oferecidos pela plataforma, que serão vistos mais adiante;
- A aplicação só é executada em resposta a alguma requisição HTTP ou através de uma tarefa que foi agendada, devendo responder em até 60 segundos.

1.2 ARMAZENAMENTO DE DADOS

O App Engine fornece quatro formas de armazenamento de dados:

- **Google Cloud Datastore:** fornece um esquema de armazenamento de dados do tipo NoSQL, com mecanismos de buscas e operações atômicas. Ele pode ser utilizado por aplicações que não serão portadas para outro tipo sistema de armazenamento, como banco de dados relacionais (como o MySQL);
- **Google Cloud SQL:** provê um serviço de banco de dados relacional, semelhante ao MySQL. Essa é uma opção a ser escolhida caso haja a possibilidade da aplicação ter de ser portada para outra plataforma que possui um sistema de armazenamento com banco de

dados relacional. É importante ter essa informação para poder decidir entre esse sistema e o Google Cloud Datastore;

- **Google Cloud Storage:** fornece um serviço de armazenamento completamente gerenciável de objetos e arquivos com tamanhos da ordem de terabytes. Ele pode ser usado como repositório de arquivos ou de websites, armazenamento de backups ou logs. Os dados armazenados nesse serviço podem ser gerenciados por uma aplicação, utilizando bibliotecas específicas, pelo console do Google Cloud Platform, ou como sistema de arquivos mapeado em um sistema operacional.
- **Blobstore:** é usado para a aplicação armazenar e fornecer objetos chamados *blobs*, que são muito maiores do que os tipos de dados que podem ser armazenados no serviço de Datastore. Pode ser utilizado em conjunto com o Google Cloud SQL ou o Datastore, para casos onde, por exemplo, a aplicação deve armazenar uma foto, vídeo ou um arquivo muito grande que não será guardado em um banco de dados.

Neste livro, será mais usado o mecanismo Google Cloud Datastore para armazenamento dos dados das aplicações de exemplo que serão criadas. Isso porque se trata de um serviço gratuito para o volume de dados que será utilizado, além de ser simples e, ao mesmo tempo, eficiente.

Obviamente, uma aplicação pode usar mais de um mecanismo de armazenamento de dados, por exemplo, utilizar o Datastore como banco de dados principal, com dados dos usuários, transações etc., e o Blobstore para fotos. A escolha entre os mecanismos de armazenamento deve levar em conta fatores como:

- **Custo de armazenamento:** o Datastore possui preços

menores e até planos gratuitos, com certos limites, em relação ao Cloud SQL;

- **Tipo do dado a ser armazenado:** a escolha entre um banco relacional ou um NoSQL deve ser levada em conta entre usar o Datastore ou o Cloud SQL;
- **Possibilidade de portar a aplicação para outras plataformas:** utilizar o Cloud SQL pode tornar a aplicação mais fácil de ser portada para outras plataformas, com pequenas ou, até mesmo, nenhuma modificação;
- **Tamanho do dado a ser armazenado:** o armazenamento de fotos e arquivos grandes requer serviços como o Blobstore ou o Cloud Storage.

1.3 AGENDAMENTO DE TAREFAS

Tarefas podem ser agendadas para serem executadas, sem a necessidade de responderem a requisições HTTP externas. Essa facilidade permite que a aplicação execute procedimentos agendados (a cada hora ou a cada dia), como requisições a recursos externos ou processos de limpeza do sistema de armazenamento.

1.4 COTAS E LIMITES

Para começar a desenvolver com o Google App Engine, basta criar uma conta e publicar a aplicação. A partir daí, qualquer pessoa poderá acessar de qualquer lugar sem nenhum custo. Porém, existem alguns limites para a utilização gratuita da plataforma, que são:

- Registrar até 10 aplicações por usuário;
- Cada aplicação pode usar até 1 GB de armazenamento de dados (NoSQL) com limite de 50 mil operações de

leitura/escrita por dia;
- 1 GB de tráfego de entrada e saída por dia.

Existem outros limites que podem ser consultados na página do Google App Engine, porém esses são os mais significativos. Ademais, o Google costuma mudar tais limites, seguindo tendências de mercado, normalmente aumentando algumas das opções gratuitas.

Você também pode habilitar o mecanismo e cobrança da sua aplicação, o que significa que nenhum serviço ou funcionalidade vai parar caso atinja o limite gratuito; mas é claro, será cobrado por isso.

1.5 CONSOLE DE ADMINISTRAÇÃO

O Google App Engine possui um console de administração Web, onde várias operações de gerenciamento das aplicações podem ser realizadas. Ele pode ser acessado no endereço https://console.developers.google.com/project.

Figura 1.1: Página inicial

Essa tela lista os projetos da sua conta no **Google Developers Console**, com algumas estatísticas. Cada um dos nomes de projeto é um link para seu *dashboard* de administração, que será detalhado ao longo dos capítulos deste livro.

Você utilizará muito esse console de administração das aplicações que forem publicadas no Google App Engine.

1.6 CONCLUSÃO

Agora que você já sabe um pouco sobre o que o Google App Engine, como plataforma, pode oferecer, você poderá preparar seu ambiente de desenvolvimento seguindo as instruções do próximo capítulo.

Capítulo 2

PREPARANDO O AMBIENTE DE DESENVOLVIMENTO

Chegou a hora de preparar seu ambiente de desenvolvimento! Para desenvolver aplicações para o Google App Engine em Java, são necessários os seguintes programas e pacotes:

- Google App Engine SDK for Java 1.9.30 - Será instalado de dentro do Eclipse.
- Eclipse IDE for Java EE Developers 4.4 Luna 64 bits - https://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/lunasr2.
- Java Development Kit 7 64 bits - http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html.
- Google Plugin for Eclipse 4.4 (Luna) - Será instalado de dentro do Eclipse.

As versões apresentadas dos programas e pacotes formam uma combinação recomendada para o Google App Engine SDK para a versão 1.9.30 e para a linguagem Java. Obviamente, se uma versão mais nova do SDK surgir, talvez possam ser usadas versões mais novas de Eclipse e/ou Java. Tente utilizar essas versões

recomendadas, pelo menos durante os exercícios do livro, para que você não tenha contratempos indesejados.

Baixe esses pacotes e siga as instruções nas próximas seções. O plugin do Google para o Eclipse será instalado de dentro do Eclipse.

2.1 INSTALANDO O JDK 7

Dê um duplo clik no arquivo baixado do Java Development Kit 7, e siga as instruções de instalação.

2.2 GOOGLE APP ENGINE SDK FOR JAVA 1.9.30

O Google App Engine SDK for Java pode ser baixado ou instalado de dentro do Eclipse, que é o que será feito um pouco mais adiante neste capítulo, pois assim tudo já ficará configurado corretamente.

O App Engine SDK possui as seguintes características:

- Todas as APIs e bibliotecas disponíveis da plataforma GAE;
- Um ambiente seguro para simular e testar suas aplicações na sua máquina local de desenvolvimento, que emula a infraestrutura do GAE, sem a necessidade de publicar a aplicação e consumir seus recursos;
- Ferramentas de desenvolvimento para subir a aplicação para o GAE pelo Eclipse.

2.3 INSTALANDO E CONFIGURANDO O ECLIPSE

Descompacte o Eclipse em uma pasta na sua máquina de desenvolvimento. Depois disso, abra-o clicando em seu executável e

execute os passos a seguir:

1. Configure o local do *workspace* que você desejar:

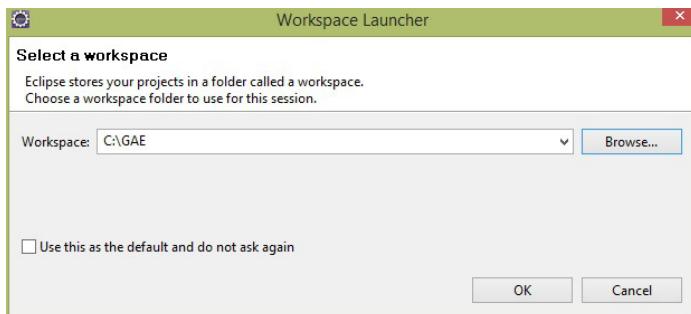



Figura 2.1: Configurando o local do workspace

2. Clique em `OK` e aguarde até que o Eclipse carregue totalmente;
3. Acesse o menu `Window -> Preference` para abrir a janela de configurações do Eclipse;
4. Nessa janela, acesse a opção `Java -> Installed JREs` ;
5. Adicione um novo JRE, clicando no botão `Add` . Selecione o local onde você instalou o JDK 7;
6. Selecione a opção `jdk1.7.0_80` como a opção padrão. Isso fará com que as novas aplicações sejam criadas utilizando essa JRE;

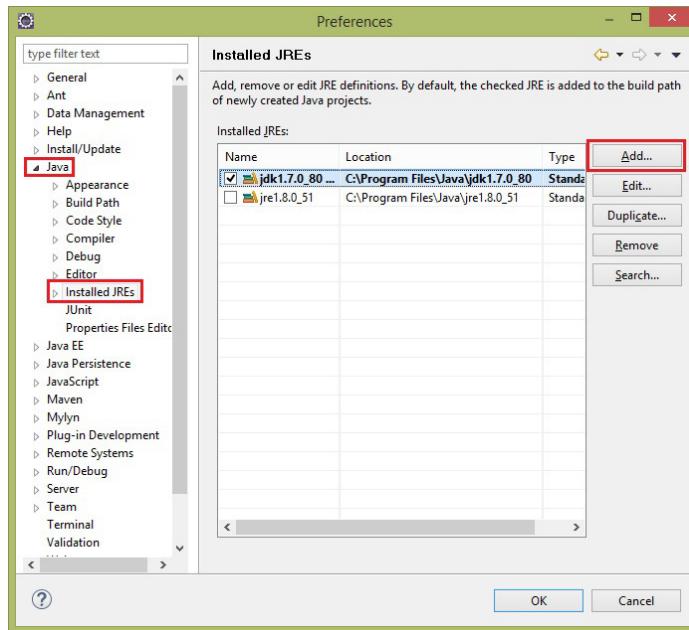



Figura 2.2: Configurando a JRE padrão

7. Clique em OK.

2.4 INSTALANDO O GOOGLE PLUGIN FOR ECLIPSE

Para instalar o Google Plugin for Eclipse, execute os passos a seguir, dentro do Eclipse:

1. Acesse o menu `Help -> Install New Software` ;
2. Na janela que abrir, clique no botão `Add` para adicionar o repositório de ferramentas do Google para o Eclipse 4.4;
3. Adicione o endereço `https://dl.google.com/eclipse/plugin/4.4` no campo

`Location` no pop-up que se abrir, e clique em `OK` ;

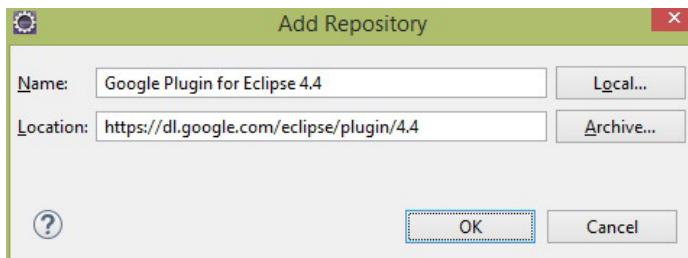



Figura 2.3: Configurando o repositório do Google Plugin for Eclipse

4. Aguarde até que o Eclipse carregue as opções de ferramentas para a instalação;
5. Depois de o Eclipse carregar as opções de instalação do repositório do Google, selecione as seguintes opções, pelo menos:
   - Google App Engine Maven Integration
   - Google Plugin for Eclipse 4.4
   - Google App Engine Java SDK 1.9.30

   A tela de instalação do Eclipse deverá ficar como a figura a seguir:

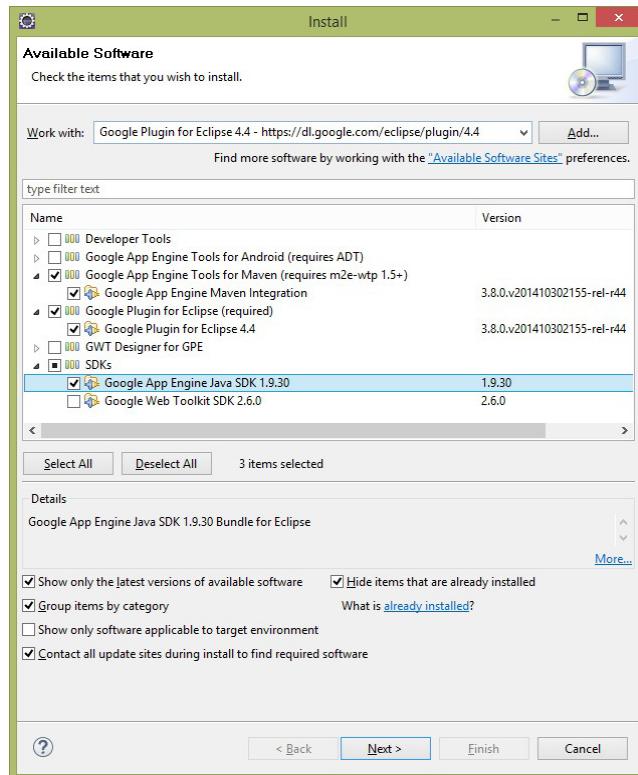



Figura 2.4: Opções de instalação no Eclipse

6. Clique em `Next` ;
7. Aceite todos os termos das licenças, e depois cliquem em `Finish` para dar início ao processo de download e instalação;
8. Depois que tudo for baixado e instalado, o Eclipse pedirá para ser reiniciado;
9. Depois de reiniciar o Eclipse, acesse o menu `Window -> Preference` e veja que há uma nova opção com o nome de `Google` ;

10. Expanda a opção `Google` e clique no item `App Engine` . Veja que o App Engine SDK já está instalado e configurado corretamente:

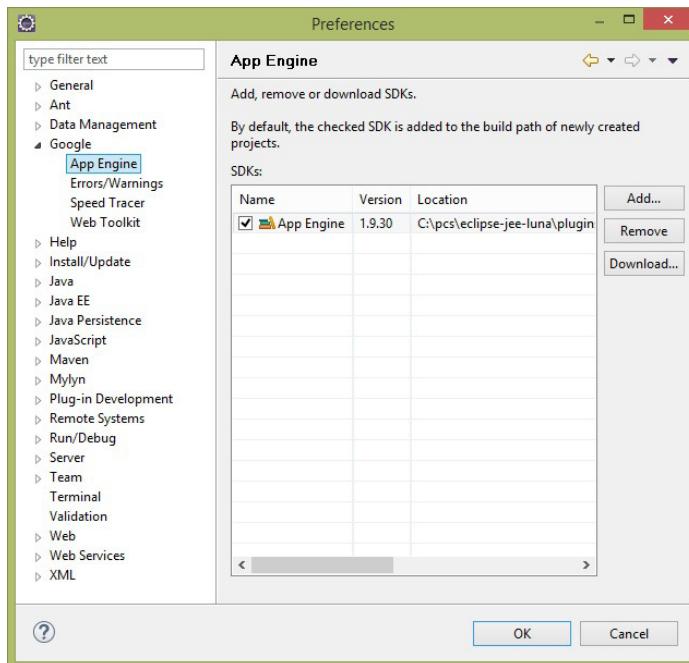



Figura 2.5: Google App Engine instalado

11. Nessa mesma tela, vá à opção `Java -> Compiler` e configure o item `Compiler compliance level` para **1.7**;

12. Repare também que, no canto inferior direito do Eclipse, há a opção para você deixar o Eclipse logado na sua conta do Google. É interessante fazer isso para que ele acesse a sua conta durante a criação de novos projetos, criando uma aplicação nova no GAE;

> **Utilizando o Maven**
>
> O plugin do Google para o Eclipse possui boas ferramentas para a criação e publicação de projetos para o GAE, mas neste livro você criará um projeto utilizando o Maven como gerenciador de dependências. Por isso, o processo será um pouco diferente, mas valerá a pena quando você tiver de acrescentar bibliotecas ao projeto, como será o caso no capítulo *Construindo serviços REST com Jersey*, onde será inserido o Jersey para trabalhar com REST.

13. Ainda na tela de preferências do Eclipse, vá à opção `Maven -> Archetypes` e clique no botão `Add Remote Catalog` para adicionar o catálogo de arquiteturas de projeto, dentre elas a de projetos para o Google App Engine;
14. Na tela que se abrir, preencha o campo `Catalog File` com o endereço `http://repo1.maven.org/maven2/archetype-catalog.xml` e clique em `OK` ;
15. Clique em `OK` para fechar a tela de preferências do Eclipse;
16. Realize as demais configurações no *workspace* do Eclipse que você desejar ou que já está acostumado.

2.5 CONCLUSÃO

Esses foram os passos para a preparação do ambiente de desenvolvimento para trabalhar com o Google App Engine. No próximo capítulo, você vai construir sua primeira aplicação em Java e publicá-la no GAE.

CAPÍTULO 3

DESENVOLVENDO A PRIMEIRA APLICAÇÃO PARA O GAE

Para entender como funciona o processo de criação de uma aplicação em Java, e depois como é feita a publicação no GAE, você construirá um primeiro projeto, chamado `exemplo1` . Nos capítulos seguintes, você vai incrementá-lo, adicionando:

- Um serviço **REST**;
- Persistência de dados no **Google Cloud Datastore**;
- Mensagens de **log**;
- **Autenticação** HTTP Basic;
- Interação com o **Google Cloud Messaging**.

Com isso, você também aprenderá técnicas interessantes de trabalhar com o console de administração do Google App Engine, como:

- Monitorar a execução de uma aplicação;
- Visualizar os acessos, com monitoramento das mensagens de erro;
- Visualizar as mensagens de log geradas pela aplicação;
- Gerenciar o Google Datastore para visualizar e até editar os dados gravados pela aplicação;
- Gerenciar as versões da aplicação;

- Configurar a aplicação para poder utilizar o Google Cloud Messaging.

3.1 CONSTRUINDO O PROJETO BÁSICO

Para começar a criar o projeto `exemplo1`, execute os passos a seguir:

1. Vá até o endereço https://console.cloud.google.com/project, e clique no botão `Create project`, para criar um novo projeto no GAE:

Figura 3.1: Criar novo projeto no GAE

2. Na tela que aparecer, escolha o nome do seu projeto. Lembre-se de que esse nome será o `Project ID`, que deverá ser associado na criação do projeto no Eclipse. Ele também deve ser único na plataforma do GAE, pois ele será parte da URL de acesso quando ele for publicado;

3. Se desejar, escolha também a região onde ele será hospedado:

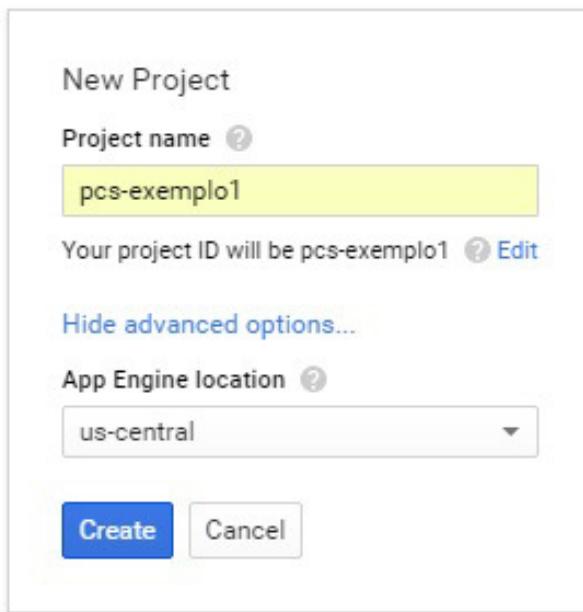



Figura 3.2: Criando novo projeto no GAE

4. Clique no botão `Create` e aguarde até que o GAE crie seu projeto;

   Assim que o projeto for criado, você será redirecionado para o *dashboard* principal da plataforma do Google Cloud. Nos capítulos adiante, você aprenderá algumas ferramentas da seção `App Engine` desse *dashboard*.

5. Tendo o `Project ID` do projeto recém-criado, vá ao Eclipse e acesse o menu `File -> New -> Maven Project` :

6. Na tela que aparecer, clique em `Next` ;

7. Na opção `Catalog` , escolha o catálogo adicionado no capítulo anterior e, em seguida, filtre por `com.google.appengine.archetypes` ;

8. Dentre as opções que aparecerem, escolha o `Artifact Id` com o nome de `appengine-skeleton-archetype`, como mostra a figura a seguir:

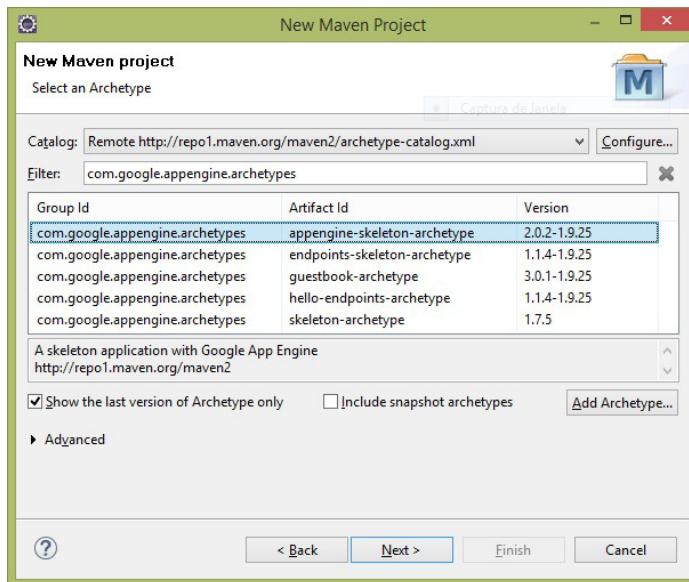



Figura 3.3: Escolha do tipo do projeto

9. Clique em `Next` e preencha os dados para a criação do novo projeto:

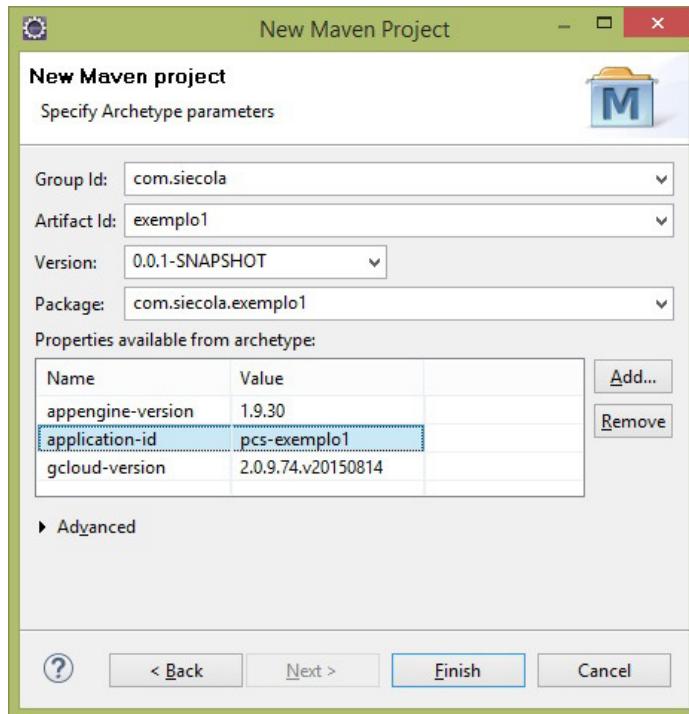



Figura 3.4: Dados para criação do projeto

Nessa tela, é importante se atentar para a versão do App Engine, que no caso foi colocado `1.9.30` e o `Application Id`, que deve ser igual ao que você criou para o novo projeto no console do GAE.

10. Clique em `Finish` para a criação do projeto e aguarde.

    Dois erros de **verificação do Eclipse** aparecerão nos arquivos `nbactions.xml` e `pom.xml`. Não é necessário tomar nenhuma atitude para que o projeto funcione, mas o Eclipse pode ficar reclamando toda vez que você for publicar o projeto. Por isso, se desejar, continue executando os passos a seguir:

11. No arquivo `nbactions.xml` , remova as três primeiras linhas desse arquivo para resolver o primeiro problema de verificação apontado pelo Eclipse, como dito no passo anterior:

```
#set( $symbol_pound = '#' )
#set( $symbol_dollar = '$' )
#set( $symbol_escape = '\' )
```

12. No arquivo `pom.xml` , vá até onde o Eclipse mostra o erro, que é em uma tag com o nome de `execution` , posicione o cursor e pressione `CTRL + 1` para abrir as opções de correção desse erro;

13. Clique na opção `Permanently mark goal...` , que aparece duas vezes. Repita o processo novamente. Isso vai resolver o segundo problema de verificação do Eclipse que, como dito nos passos anteriores, apenas indica que ele encontrou algo de errado. Deixar esse erro pode ser um pouco irritante toda vez que você for colocar a aplicação para rodar.

14. Clique no projeto e acesse o menu `Maven -> Update Project` . Certificar-se de que seu projeto está completamente atualizado com todas as dependências que o arquivo `pom.xml` do Maven diz para ele ter. Seu projeto agora deve ficar sem nenhum erro.

Nesse momento, o projeto apenas foi criado e vinculado à aplicação criada no GAE, porém ainda não foi feito o deploy.

Estrutura do projeto

A estrutura de diretórios e arquivos do projeto criado deverá ficar como mostra a figura a seguir:

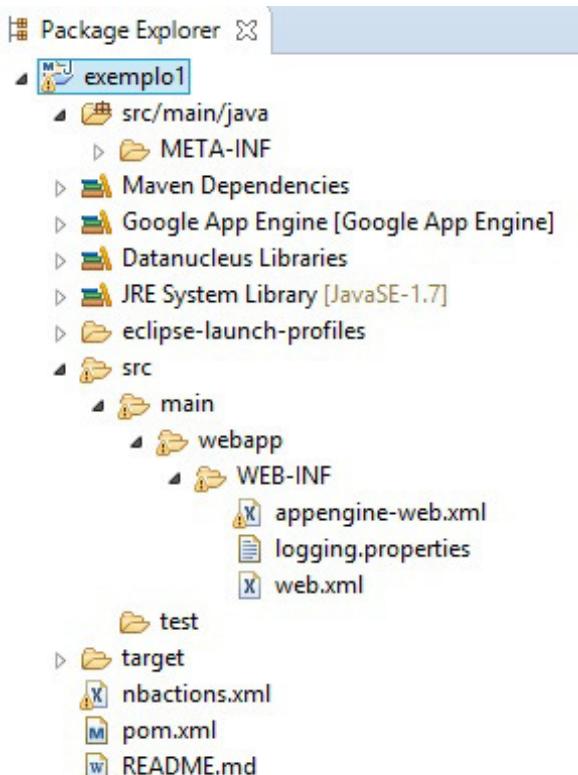



Figura 3.5: Estrutura do projeto

Dentro da estrutura do projeto, há vários arquivos que foram criados pelo Eclipse. No momento, os que são de interesse são:

- O arquivo `src/main/webapp/WEB-INF/web.xml` : nele serão feitas as configurações de qual classe de servlet será chamada quando uma requisição for recebida;

- O arquivo `src/main/webapp/WEB-INF/appengine-web.xml` : nele estão as configurações de *deploy and run* da aplicação.

Outros arquivos, de classes e/ou de configurações, serão

adicionados ao projeto à medida que ele for ganhando novas funcionalidades.

3.2 ACRESCENTO O PRIMEIRO SERVLET

Para testar a aplicação, antes é necessário adicionar um servlet para exibir uma mensagem de boas-vindas. O template do projeto criado não possui nada desse tipo, por isso, execute os passos a seguir:

1. No projeto `exemplo1`, na pasta `src/main/webapp`, crie o arquivo `index.html` para ser o ponto de entrada da aplicação, com um link para o servlet que será criado a seguir:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//
EN">
<html>
  <head>
    <meta http-equiv="content-type" content="text/html; cha
rset=UTF-8">
    <title>Hello App Engine</title>
  </head>
  <body>
    <h1>Hello App Engine!</h1>
    <table>
      <tr>
        <td colspan="2" style="font-weight:bold;">Available
 Servlets:</td>
      </tr>
      <tr>
        <td><a href="exemplo1">Exemplo1</a></td>
      </tr>
    </table>
  </body>
</html>
```

2. Crie o pacote `com.siecola.exemplo1`, onde o servlet será criado;

3. Dentro desse pacote, crie a classe `Exemplo1Servlet`. Nele será criado o método que vai tratar a requisição ao servlet:

```
import javax.servlet.http.HttpServlet;

@SuppressWarnings("serial")
public class Exemplo1Servlet extends HttpServlet {

}
```

4. Crie o método `doGet`, que será invocado quando o usuário clicar no link da página `index.html` e invocar esse servlet:

```
public void doGet(HttpServletRequest req, HttpServletRespon
se resp)
        throws IOException {
    resp.setContentType("text/plain");
    resp.getWriter().println("Hello, world");
}
```

5. Abra o arquivo `web.xml` e acrescente as configurações a seguir para adicionar o caminho do novo servlet e também a página `index.html` como sendo o arquivo de boas-vindas:

```
<servlet>
    <servlet-name>Exemplo1</servlet-name>
    <servlet-class>com.siecola.exemplo1.Exemplo1Servlet</se
rvlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Exemplo1</servlet-name>
    <url-pattern>/exemplo1</url-pattern>
</servlet-mapping>
<welcome-file-list>
    <welcome-file>index.html</welcome-file>
</welcome-file-list>
```

Agora, sua aplicação já está pronta para ser executada na sua máquina local de desenvolvimento!

3.3 EXECUTANDO A APLICAÇÃO

A aplicação criada na seção anterior pode ser executada localmente para fins de depuração como uma aplicação Web comum construída em Java. Para isso, execute os passos a seguir:

1. Clique com o botão direito no projeto e acesse o menu `Run As --> Run on Server`. Isso fará com que as opções de execução da aplicação sejam exibidas;
2. Selecione a opção `Google -> Google App Engine` para escolher executar a aplicação no ambiente de desenvolvimento simulado da sua máquina, como vemos na figura a seguir:

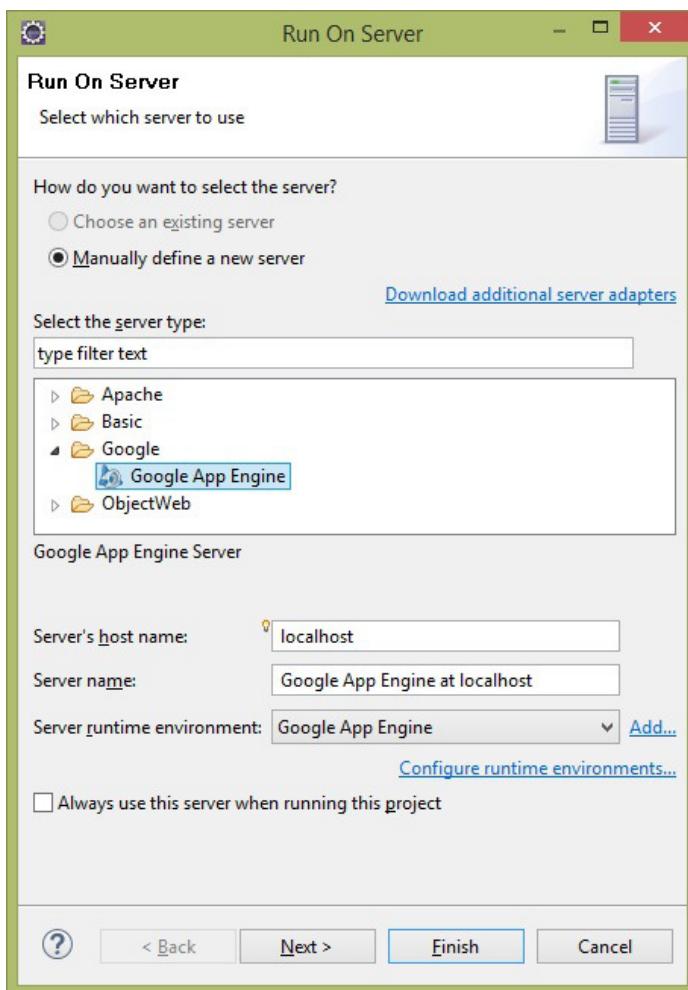



Figura 3.6: Executando o projeto

3. Clique em `Finish` para que o Eclipse crie um novo servidor na aba `Servers`, e adicione o projeto `exemplo1` para ser executado nele.

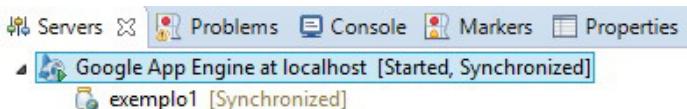



Figura 3.7: Projeto na aba Servers

Um servidor será iniciado para que a aplicação seja executada em um ambiente simulado, semelhante à infraestrutura disponível no GAE.

Nesse momento, o Eclipse abrirá uma aba no endereço http://localhost:8888/, onde a página `index.html` criada está sendo exibida. Você também pode acessar essa página pelo browser da sua máquina.

Figura 3.8: Página de boas-vindas

A página exibirá o link para o servlet que foi criado. Clique nele para acessá-lo. Isso redirecionará a requisição para o servlet que foi criado, de nome `Exemplo1Servlet`.

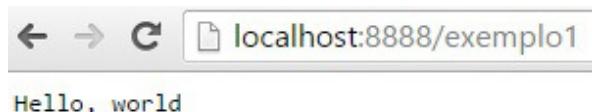



Figura 3.9: Hello World

Executar e depurar a aplicação na máquina local de desenvolvimento é muito interessante, pois é muito mais simples de encontrar os erros no código, uma vez que é possível executá-lo passo a passo, como em um programa em Java normal.

Há também um console de administração local, que pode ser acessado em http://localhost:8888/_ah/admin.

Figura 3.10: Console de administração local

Algumas dessas funções, como o `Datastore Viewer` , poderão ser utilizadas no capítulo *Armazenando dados no Google Cloud Datastore*, para visualizar as entidades e os dados armazenados na sua máquina local de desenvolvimento.

3.4 PUBLICANDO NO GAE

Para colocar a aplicação para rodar no GAE, clique com o botão direito sobre o servidor criado na aba `Servers` do Eclipse, e acesse o menu `Google App Engine WTP -> Deploy to Remote Server`.

Aguarde até o processo finalizar. Abra seu browser padrão apontando para endereço da aplicação criada, rodando no GAE com o código criado. No exemplo que está sendo criado, a URL de acesso do projeto publicado é a seguinte: `https://pcs-exemplo1.appspot.com/`

Com certeza a URL do seu projeto será diferente, pois o `Project ID` que você criou é outro. Esse processo pode ser feito sempre que uma alteração ou correção for feita na aplicação e necessitar ser atualizada para que outras pessoas ou sistemas possam acessar.

3.5 EXPERIMENTANDO O CONSOLE DO GAE

O console de administração do GAE, que pode ser acessado em https://console.cloud.google.com/project, oferece várias ferramentas de configuração e administração das aplicações publicadas nele. Aqui, algumas delas:

- Um *dashboard* com as principais informações gerais sobre a aplicação selecionada, com um gráfico de requisições;
- Informações sobre as versões publicadas;
- Informações sobre as instâncias em execução;
- Quais tarefas estão agendadas;
- Detalhes de utilização dos recursos da plataforma e as cotas disponíveis.

À medida que você for estudando os capítulos do livro, você vai

aprender mais sobre o console de administração do GAE e como ele pode ajudar.

3.6 CONFIGURANDO A VERSÃO DA APLICAÇÃO

Uma configuração importante é a versão da aplicação, que pode ser feito na *tag version* do arquivo `appengine-web.xml` , como no exemplo a seguir:

```
<version>1-1</version>
```

Com isso, é possível manter várias versões em execução ao mesmo tempo no App Engine, e ainda poder escolher qual é a versão que será acessada por padrão, como se vê na figura seguinte. Isso pode ser acessado no menu principal, na seção `Versions` do App Engine:

Versions   MAKE DEFAULT   DELETE

Version	Size	Runtime	Instances
1-1	3.3 MB	java7	0
1 (default)	3.3 MB	java7	0

Figura 3.11: Versões da aplicação

Nessa página, pode-se observar qual versão é a padrão - aquela que será executada se o usuário acessar a URL da aplicação, sem especificar qual versão deseja. As demais versões da aplicação em execução podem ser acessadas colocando sua identificação antes da sua URL, como no exemplo a seguir. Da mesma forma, os relatórios de uso, gráficos e logs podem ser filtrados para cada versão: `http://1-1-dot-pcs-exemplo1.appspot.com/`

Ainda na página de controle de versões, é possível tornar uma versão padrão ou mesmo apagar alguma que você não deseja que fique disponível mais.

Apenas lembrando, o código desse projeto está no repositório do livro, no endereço: https://github.com/siecola/GAEBook

3.7 CONCLUSÃO

Neste capítulo, você aprendeu a publicar seu primeiro projeto no Google App Engine! Além disso, pode conhecer um pouco do seu console de administração Web, uma ferramenta que pode ajudar muito no gerenciamento das aplicações publicadas no GAE.

No próximo capítulo, você verá como criar um serviço REST nesse primeiro projeto, utilizando o framework Jersey.

CAPÍTULO 4

CONSTRUINDO SERVIÇOS REST COM JERSEY

A API Jersey é a implementação de referência da especificação JAX-RS (JSR 339) para construção de Web Services RESTful. Com ela, é possível construir uma aplicação no GAE para prover serviços REST de forma descomplicada, como se estivéssemos trabalhando em um projeto como Apache TomCat ou outro servidor.

Para trabalhar com a API Jersey, é necessário adicionar algumas bibliotecas ao projeto, como será descrito na próxima seção.

4.1 PREPARANDO O PROJETO PARA TRABALHAR COM SERVIÇOS REST

Nesta seção será demonstrado como preparar o projeto `exemplo1` que rodará no GAE com a API Jersey, Datastore, JSON e Google Cloud Messaging, a serem mostrados nos capítulos seguintes. Para isso, execute os passos a seguir, para adicionar as dependências ao projeto:

1. Abra o arquivo `pom.xml` , localizado na raiz da projeto, que contém as suas configurações de dependências;

2. Procure pela tag `dependencies` , onde as dependências são listadas;

3. Dentro dessa tag, acrescente a dependência do Jersey, que será responsável, dentre outras coisas, pela criação dos serviços REST:

```
<dependency>
    <groupId>org.glassfish.jersey.containers</groupId>
    <artifactId>jersey-container-servlet</artifactId>
    <version>2.22.1</version>
</dependency>
```

4. Ainda dentro dessa tag, acrescente a dependência da biblioteca Jackson para trabalhar com o formato JSON com o Jersey:

```
<dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-json-jackson</artifactId>
    <version>2.22.1</version>
</dependency>
```

   Existem outras bibliotecas para se trabalhar com JSON com o Jersey, mas essa é uma das mais conhecidas e fáceis de se trabalhar. Você verá no próximo capítulo que os métodos poderão ser construídos com parâmetros de retorno do tipo de objetos complexos e até listas deles, sem a necessidade de chamar qualquer método para convertê-los para JSON - o que acelera muito o desenvolvimento.

1. Salve o arquivo `pom.xml`, que o Maven vai se encarregar de baixar os arquivos `.jar` para utilizar o Jersey com todas as suas dependências.

Agora que o projeto já possui as bibliotecas e dependências básicas, é possível começar a desenvolver o primeiro serviço REST com Jersey.

4.2 CRIANDO O PRIMEIRO SERVIÇO REST

Aqui será criado um serviço REST simples, para que o ambiente possa ser validado com as configurações mínimas. Para isso, execute os passos a seguir:

1. No Eclipse, crie o pacote `com.siecola.exemplo1.services` na aplicação `exemplo1`. Nesse pacote é que serão criadas as classes que vão implementar os serviços REST. É recomendado que todas as classes fiquem em um mesmo pacote, pois assim facilita a configuração de mapeamento do Jersey a ser realizado no arquivo `web.xml`;

2. Abra o arquivo `web.xml` da aplicação, localizado na pasta `src/main/webapp/WEB-INF`. Ele contém configurações que são verificadas na inicialização da aplicação, como os servlets existentes, que dizem ao Jersey onde estão os serviços REST que devem ser providos e parâmetros para serem configurados nesse momento;

3. Antes do fechamento da tag `web-app`, adicione o trecho a seguir. Ele diz ao Jersey que os serviços estão no pacote `com.siecola.exemplo1.services` e também que o caminho padrão para acessá-los deverá ser `/api/`:

```
<servlet>
    <servlet-name>Exemplo1Services</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletCont
ainer</servlet-class>
    <init-param>
        <param-name>jersey.config.server.provider.packages</
param-name>
        <param-value>com.siecola.exemplo1.services</param-v
alue>
    </init-param>
    <init-param>
        <param-name>org.glassfish.jersey.api.json.POJOMappi
ngFeature</param-name>
        <param-value>true</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

```
<servlet-mapping>
    <servlet-name>Exemplo1Services</servlet-name>
    <url-pattern>/api/*</url-pattern>
</servlet-mapping>
```

Isso fará com que todos os serviços anotados na classe `com.siecola.exemplo1.services` sejam tratados como serviços REST a serem expostos. Por isso, é importante criar um único pacote, onde devem ficar todas as classes que vão implementar um serviço REST.

4. No pacote `com.siecola.exemplo1.services`, crie a classe `HelloWorld`, que implementará o primeiro serviço criado:

```
package com.siecola.exemplo1.services;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;

@Path("/helloworld")
public class HelloWorld {

    @GET
    @Produces("application/json")
    @Path("teste/{name}")
    public String helloWorld(@PathParam("name") String name
) {

        String res = "Hello World " + name;

        return res;
    }
}
```

Essa é a implementação do primeiro serviço com a sua primeira operação, que possui apenas um método (`helloWorld`), e que recebe a `string name` como parâmetro e retorna uma outra `string` concatenada com ela. Porém, é necessário ressaltar alguns pontos importantes, como:

- A anotação `@Path` no início da definição da classe

indica que todas as operações que estiverem nessa classe serão acessadas por essa `URL`, que na verdade será adicionada ao que já estava na configuração do `web.xml`, ficando, para esse exemplo, `/api/helloworld`;

- A anotação `@GET` indica que o método será acessado pela operação `GET` do HTTP. Nessa classe, todos os métodos públicos com anotações dos verbos HTTP serão considerados operações do serviço;
- `@Produces` indica que a resposta será no formato especificado, nesse caso `application/json`;
- A anotação `@Path` no início da implementação do método `helloWorld` indica o restante da URL para que essa operação possa ser acessada, que nesse exemplo ficará `/api/helloworld/teste/{name}` - sendo a última parte (`{name}`) usada como parâmetro do método, por meio da anotação `@PathParam`.

Mais adiante, outras anotações serão apresentadas e também métodos mais sofisticados serão criados para construção de operações elaboradas, contendo objetos complexos como parâmetros de entrada e/ou saída. Isso tudo vem da especificação JAX-RS (JSR 339), aqui implementada pela API Jersey.

4.3 TESTANDO O SERVIÇO

Para testar o serviço `helloworld` na máquina local de desenvolvimento, sem publicar no GAE, execute os seguintes passos:

1. Execute a aplicação no Eclipse;
2. Abra um browser e acesse o endereço da operação do serviço `helloworld`, que deverá ser

`http://localhost:8888/api/helloworld/teste/Matilde` , sendo que a última parte do endereço (`Matilde`) é o parâmetro a ser passado para a operação. O resultado deve ser como mostra a figura a seguir:

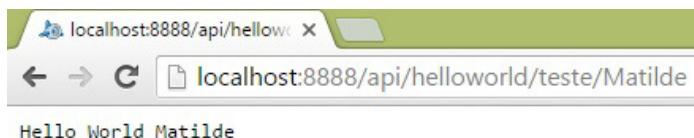



Figura 4.1: Acessando o serviço

O comportamento da operação, como descrito, é retornar a frase de boas-vindas com o nome passado como parâmetro.

4.4 PUBLICANDO A APLICAÇÃO NO GAE

Agora que a aplicação `exemplo1` possui um serviço REST, no projeto do Eclipse, altere a versão dela no arquivo `web.xml` para `1-1` , e publique no Google App Engine.

Vá ao console do GAE no menu `Versions` , e altere a versão padrão para `1-1` , como mostra a figura a seguir, para que você possa ver a nova versão publicada. Para isso, clique na versão `1-1` e depois no botão `Make Default` :

Versions MAKE DEFAULT DELETE

Version	Size	Runtime	Instances
1-1 (default)	3.3 MB	java7	0
1	3.3 MB	java7	0

Figura 4.2: Alterando a versão padrão

Acesse o serviço `helloworld` algumas vezes e depois visualize o gráfico de requisições no console do GAE, na seção `Dashboard` . Repare que a versão `1-1` já aparece selecionada para exibir os dados no gráfico:

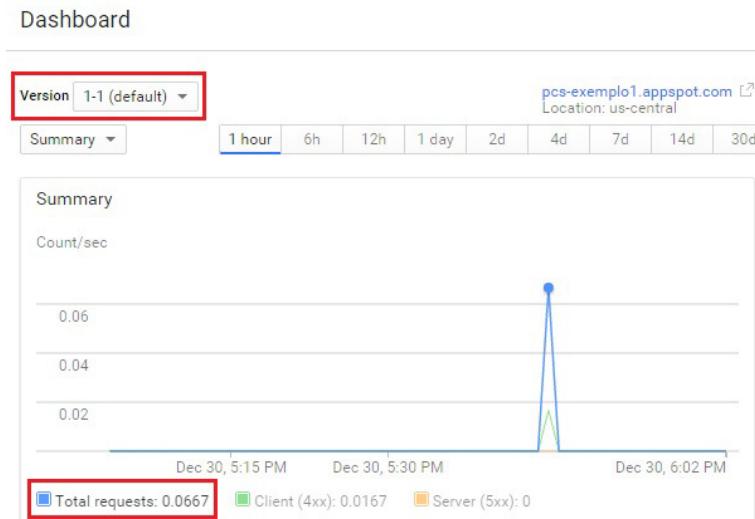



Figura 4.3: Gráfico de requisições

Nessa mesma página, é possível ver os acessos à aplicação por URL:

Current load

URI	Requests/Minute Current	Requests Last 24 hours	Runtime MCycles Last hour	Average latency Last hour
/api/helloworld/teste/matilde	0.4	0	185	179 ms
/exemplo1	0.2	0	12,109	8,514 ms

Figura 4.4: Acessos à aplicação

4.5 TESTANDO O SERVIÇO COM O REST CONSOLE

Embora o Google Chrome possua boas opções para o

desenvolvedor, é necessário utilizar uma ferramenta que permita customizar valores de campos nas requisições bem como escolher qual o método HTTP a ser usado - como `PUT` , `POST` ou `DELETE` . Para isso, será utilizado o plugin chamado REST Console.

Para instalá-lo, basta acessar a URL http://www.restconsole.com, que o Google Chrome será redirecionado para a sua loja, onde será possível instalar o plugin. Depois de instalado, basta acessá-lo na página de extensões do seu Chrome. A figura a seguir mostra uma parte desse plugin aberto e pronto para ser utilizado:

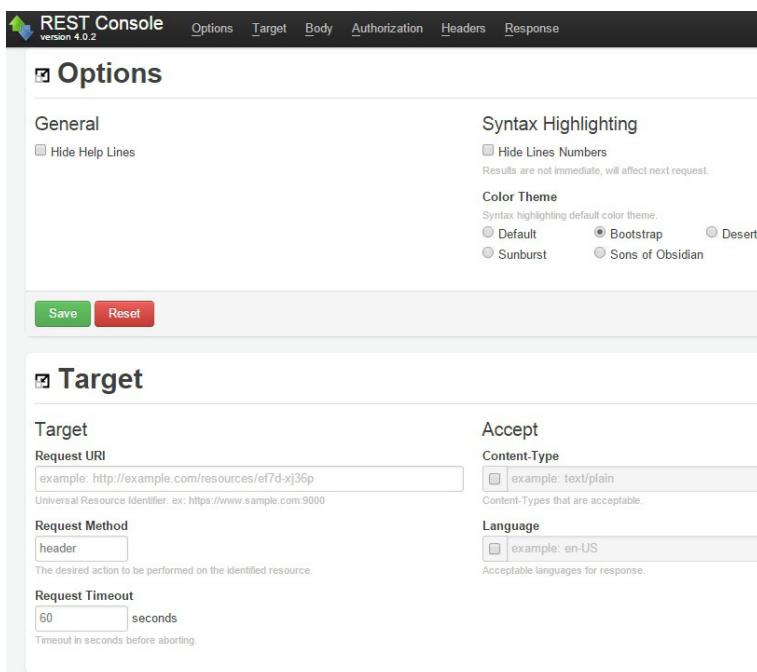



Figura 4.5: REST Console

O REST Console possui as seguintes seções:

- **Options:** permite a configuração de aparência do plugin;

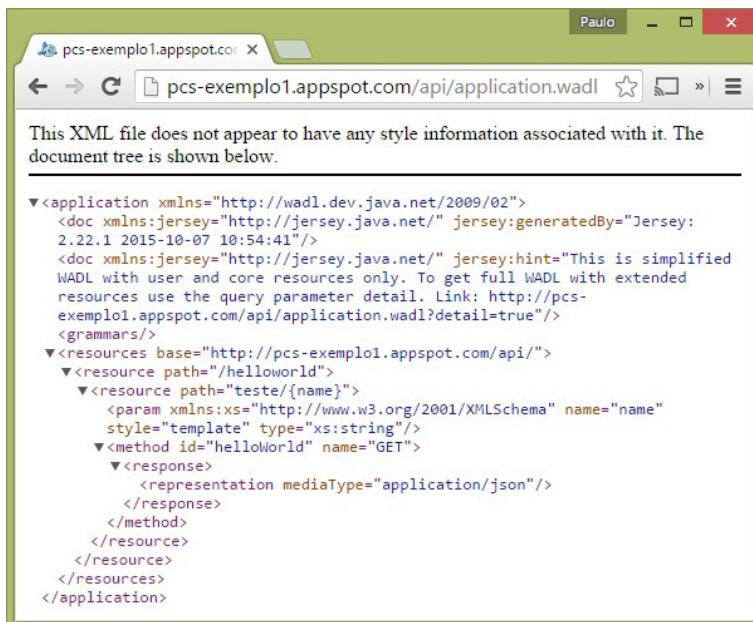
- **Target:** é onde o endereço da requisição deverá ser preenchido, além de outras configurações, como valor do campo `Accept`, que permite definir o tipo de dado desejado na resposta, método HTTP da requisição e timeout;
- **Body:** permite que o usuário escolha o formato do dado que será enviado no campo `Request Payload`;
- **Authorization:** permite a configuração de parâmetros para requisições que exijam autenticação;
- **Headers:** permite que o usuário configure alguns cabeçalhos da requisição HTTP;
- **Botões de comandos:** onde o usuário pode selecionar qual o método HTTP será usado na requisição. O botão `Send` realiza a requisição com o método definido no campo `Request Method` da seção `Target`;
- **Response:** onde será exibida a resposta da requisição à operação do serviço. Nessa seção é possível verificar campos da requisição, da resposta, bem como seu conteúdo.

A figura a seguir mostra como o REST Console pode ser configurado para acessar a operação `helloworld`:

Figura 4.6: Configurando o REST Console

Dessa forma, ao clicar no botão `GET` do REST Console, ele exibirá na seção `Response` a resposta à consulta a `helloworld` , como vemos adiante:

Figura 4.7: Resposta do REST Console

Porém, ainda é possível realizar outras análises nas abas da seção `Response` do REST Console, como:

- Verificar o formato original dos dados de resposta;
- Visualizar os cabeçalhos da mensagem de resposta;
- Analisar o corpo da mensagem de requisição;
- Visualizar os cabeçalhos da mensagem de requisição.

> O REST Console será amplamente utilizado ao longo deste livro para acessar as operações dos serviços que forem sendo criados. É interessante habituar-se com ele desde já.

4.6 GERAÇÃO DO CONTRATO DO SERVIÇO COM WADL

Com o Jersey, é possível gerar o WADL (*Web Application Description Language*) dos serviços, que é uma nova forma para contratos de serviços REST, bem difundida e com suporte de várias ferramentas e IDEs.

Para obtê-lo, basta acessar a URL `/api/application.wadl` , que o arquivo será retornado, como na figura a seguir:

Figura 4.8: WADL do serviço

Nele é possível observar que:

- Os recursos (ou serviços nesse caso) estão sob o domínio `http://pcs-exemplo1.appspot.com/api/` ;
- Há uma lista de recursos, mas que nesse caso há somente um, definido com o caminho `/helloworld` , que é o serviço criado;
- Há uma operação dentro desse serviço e que pode ser acessada pelo caminho `teste/{name}` ;

- Essa operação de nome `helloWorld` deve ser acessada com `HTTP GET`, e retorna um objeto do tipo `application/json`.

Esse arquivo pode ser utilizado para a geração automática de código dos clientes para acessar o serviço descrito por ele, assim como acontece com o WSDL (***W**eb **S**ervice **D**efinition **L**anguage*) nos serviços do tipo SOAP (***S**imple **O**bject **A**ccess **P**rotocol*).

Um serviço SOAP (*Simple Object Access Protocol*) é uma forma diferente de implementação de serviços Web. Um irmão mais velho do REST, pode-se assim dizer. Os modelos de dados e métodos que são providos são definidos no WSDL. Todas as mensagens são trafegadas utilizando XML, também por meio de requisições HTTP.

Para maiores informações sobre como trabalhar com a geração de WADL com o Jersey, consulte: https://jersey.java.net/documentation/latest/wadl.html

4.7 CONCLUSÃO

Neste capítulo, você aprendeu:

- Como preparar o projeto para trabalhar com serviços REST, utilizando o **Jersey**;
- Criar um **serviço REST simples**;
- Conceitos importantes da especificação JAX-RS (JSR 339) e como anotar uma classe e seus métodos para que possam ser entendidos como serviços, operações, com seus caminhos de acesso e parâmetros;
- Um pouco sobre o console do GAE;
- Como utilizar o **REST Console** para acessar um serviço REST;
- Como gerar o contrato do serviço com o **WADL**.

No próximo capítulo, você criará um serviço um pouco mais complexo, com várias operações e com um modelo de dados com vários atributos. Dessa forma, você aprenderá mais sobre como trabalhar com o Jersey e um pouco mais sobre o console do GAE.

Capítulo 5

CRIANDO UM SERVIÇO REST COMPLETO COM JERSEY

No capítulo anterior, você criou seu primeiro serviço REST e publicou no GAE. Ele tinha apenas uma operação, que recebia uma `string` como parâmetro e retornava outra, concatenada com o parâmetro de entrada. Foi simples, porém útil para testar o projeto e começar a aprender alguns conceitos.

Este capítulo explica a criação de serviços REST um pouco mais complexos no GAE, com a utilização da API Jersey. O que será criado é um serviço de gerenciamento de produtos, mas ainda sem persistência dos dados, para que o foco fique somente na parte da criação do serviço em si. Mais adiante, será mostrado como usar o Datastore do GAE.

O serviço de gerenciamento de produtos terá as seguintes operações com os respectivos endereços de acesso:

- GET `/api/product/{code}` : para recuperar um produto de código `{code}` ;
- GET `/api/product` : para recuperar a lista de todos os produtos;
- POST `/api/product` : para inserir um novo produto;
- PUT `/api/product/{code}` : para alterar um produto

existente, passando seu código na URL e as alterações no corpo da requisição;

- DELETE `/api/product/{code}` : para apagar o produto com o código passado na URL.

Como pode ser visto, o serviço será o que se chama de CRUD (***C**reate, **R**ead, **U**pdate and **D**elete*) de produtos.

O formato de dados a ser utilizado é o JSON, muito difundido para utilização com serviços REST, por ser mais leve e de fácil interpretação.

> **FORMATO JSON**
>
> Se você não está acostumado com o formato JSON, acesse o site (http://json.org/) para aprender como os dados são representados.
>
> Um outro site interessante para ajudar na compreensão de dados em formato JSON e também para ajudar a montar representações, principalmente de objetos complexos, é o Online JSON Viewer (http://jsonviewer.stack.hu/).

5.1 CRIANDO O MODELO DE PRODUTOS

Para criar o serviço e as operações listadas, é necessário primeiramente criar o modelo de produtos. Para isso, execute os passos a seguir:

1. Crie um novo pacote na aplicação `exemplo1` , com o nome de `com.siecola.exemplo1.models` .

   Nesse pacote, serão criados todos os modelos que serão

usados pelos serviços da aplicação. A organização desses modelos em único pacote é puramente para fins de organização da estrutura do código do projeto. Não existe nenhuma implicação em criar as classes dos modelos em pacotes diferentes, diferentemente das classes dos serviços que devem ser criados todos em um único pacote.

2. Dentro desse novo pacote, crie a classe do modelo de produtos, com o nome de `Product` .

3. Na classe `Product` , crie seus atributos e os `getters` e `setters` , como no trecho de código a seguir:

```
import java.io.Serializable;

public class Product implements Serializable {

    private static final long serialVersionUID = 1L;

    private String productID;
    private String name;
    private String model;
    private int code;
    private float price;

    //getters and setters
}
```

Esse será o modelo do produto a ser utilizado pelo serviço de gerenciamento de produtos. Todas as operações usarão essa classe, logo, uma alteração aqui afeta diretamente a compatibilidade desse serviço.

No capítulo seguinte, *Armazenando dados no Google Cloud Datastore*, o mesmo modelo de produto será utilizado para persistir suas informações no Datastore.

5.2 CRIANDO A CLASSE DO SERVIÇO

Agora, é necessário criar o serviço de gerenciamento de produtos. Para isso, execute os passos a seguir:

1. No pacote `com.siecola.exemplo1.services` , crie a classe `ProductManager` para implementar o novo serviço de CRUD de produtos.

2. Coloque a anotação `@Path` na declaração da classe, para indicar que esse serviço deverá ser acessado pelo caminho informado, como no trecho a seguir:

```
import javax.ws.rs.Path;

@Path("/products")
public class ProductManager {

}
```

Isso fará com que o serviço de gerenciamento de produtos seja acessado pela URL `/api/products` .

3. Crie o método privado `createProduct` , a ser utilizado somente para criar um produto *dummy* a partir de seu código, pois até o momento não há uma tabela onde os produtos serão armazenados. A ideia aqui é apenas mostrar como criar as operações e interagir com objetos complexos como parâmetros de entrada e saída.

```
private Product createProduct (int code) {
    Product product = new Product();
    product.setProductID(Integer.toString(code));
    product.setCode(code);
    product.setModel("Model " + code);
    product.setName("Name " + code);
    product.setPrice(10 * code);
    return product;
}
```

4. Crie o método `getProduct` para implementar a operação de ler um produto pelo seu código, passado como parâmetro:

```
@GET
@Produces(MediaType.APPLICATION_JSON)
@Path("/{code}")
public Product getProduct(@PathParam("code") int code) {

    return createProduct(code);
}
```

Essa operação de recuperar um produto específico, implementada pelo método `getProduct`, será acessada pela URI `/api/product/{code}`. O parâmetro `{code}`, passado na URL, representa o código do produto. O retorno dessa operação é um objeto complexo do tipo `Product`, e será representado no formato JSON, como no trecho a seguir:

```
{
    "productID": "3",
    "name": "Nome 3",
    "model": "Model 3",
    "code": 3,
    "price": 30.0
}
```

5. Crie a operação para retornar todos os produtos em uma lista, como:

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public List<Product> getProducts() {
    List<Product> products = new ArrayList<>();
    for (int j = 1; j <= 5; j++) {
        products.add(createProduct(j));
    }
    return products;
}
```

Essa operação, que não recebe nenhum parâmetro, pode ser acessada pela URL `api/products` com o método `HTTP GET`. Ela devolve uma lista de produtos criados no momento de sua chamada. O formato devolvido será uma lista no formato JSON, semelhante ao exemplo a seguir:

```
[{
```

```
        "productID": "1",
        "name": "Name 1",
        "model": "Model 1",
        "code": 1,
        "price": 10.0
    }, {
        "productID": "2",
        "name": "Name 2",
        "model": "Model 2",
        "code": 2,
        "price": 20.0
    }, {
        "productID": "3",
        "name": "Name 3",
        "model": "Model 3",
        "code": 3,
        "price": 30.0
    }]
```

6. Crie a operação para inserir um novo produto. Ela deve receber o produto como parâmetro de entrada e devolvê-lo como saída:

```
 @POST
 @Produces(MediaType.APPLICATION_JSON)
 @Consumes(MediaType.APPLICATION_JSON)
 public Product saveProduct(Product product) {
     product.setProductID(Integer.toString(product.getCode()
));
     return product;
 }
```

A anotação `@Consumes(MediaType.APPLICATION_JSON)` significa que a operação deve receber, através do corpo da requisição, um objeto no formato JSON. Essa operação deve ser acessada pela URL `api/products`, com o método `HTTP POST` e com o produto a ser inserido no corpo da requisição. Isso pode ser feito com o REST Console configurado como na figura a seguir, que mostra a seção `Target`:

Figura 5.1: Inserindo um novo produto com o REST Console - Target

E na seção `Body` , o REST Console deve ser configurado com na figura:

Figura 5.2: Inserindo um novo produto com o REST Console - Body

O modelo e produto a ser inserido deve ser como mostra o trecho adiante:

```
{
    "name": "Name 6",
    "model": "Model 6",
    "code": 6,
    "price": 60.0
}
```

7. Agora crie a operação para apagar um produto, como no trecho a seguir:

```
@DELETE
@Produces(MediaType.APPLICATION_JSON)
@Path("/{code}")
public String deleteProduct(@PathParam("code") int code) {
    return "Product " + code + " deleted";
}
```

Essa operação deve ser acessada pela URL `api/products` com o verbo `HTTP DELETE`.

8. Por último, crie a operação de alterar um pedido, que recebe o código como parâmetro na URL e o produto com as informações a serem alteradas no corpo da requisição. Ela retorna o produto alterado como resposta:

```
@PUT
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
@Path("/{code}")
public Product alterProduct(@PathParam("code") int code, Pr
oduct product) {
    product.setName("Nome alterado");
    return product;
}
```

Essa operação deve ser acessada pela URL `api/products/{code}` com o verbo `HTTP PUT`, passando o código do produto a ser alterado no parâmetro `{code}` da URL e o produto em si com suas alterações no corpo da requisição, em formato JSON.

Como dito anteriormente, esse serviço de gerenciamento não persiste as informações em lugar nenhum, até o momento. Mas no próximo capítulo, será mostrado como isso pode ser feito utilizando o Google Cloud Datastore.

Em resumo, as novas operações com suas URLs de acesso são:

- GET `/api/product/{code}`: para recuperar um produto de código `{code}`;

- GET `/api/product` : para recuperar a lista de todos os produtos;
- POST `/api/product` : para inserir um novo produto;
- PUT `/api/product/{code}` : para alterar um produto existente, passando seu código na URL e as alterações no corpo da requisição;
- DELETE `/api/product/{code}` : para apagar o produto com o código passado na URL.

5.3 CONCLUSÃO

Neste capítulo, você aprendeu como criar um serviço de CRUD de produtos, com um modelo de dados complexo, usando a API do Jersey. Com isso, aprendeu as anotações que podem ser utilizadas para definição do caminho e dos verbos HTTP a serem usados para acessar as operações do serviço. Também viu o formato de dados esperado como parâmetro de entrada e o que será devolvido como resposta.

No próximo capítulo, você aprenderá como utilizar o Google Cloud Datastore para criar uma tabela de produtos e poder persisti-los, em vez de utilizar uma lista em memória.

CAPÍTULO 6

ARMAZENANDO DADOS NO GOOGLE CLOUD DATASTORE

Os serviços de gerenciamento de produtos criado no projeto de `exemplo1` demonstrou como é possível trabalhar com objetos complexos, como parâmetros de entrada e saída das operações de um serviço REST. Isso é algo importante de ser bem entendido, mas ele não utilizou nenhum mecanismo de persistência de dados. A não ser que você esteja construindo um serviço que apenas realiza cálculos ou consulta outro serviço, realizar operações com banco de dados para salvar estados ou entidades é quase sempre necessário.

A plataforma do App Engine possui algumas formas diferentes de se persistir dados, que variam de acordo com tamanhos dos dados, custo de armazenamento, desempenho e compatibilidade com outros sistemas de armazenamento.

6.1 O QUE É O GOOGLE CLOUD DATASTORE

Este capítulo foca no Google Cloud Datastore, um banco de dados NoSQL com grande desempenho e altamente escalável. Aqui vão algumas características importantes desse serviço de armazenamento do Google:

- Transações atômicas;

- Alta disponibilidade de leituras e escritas;
- Alto desempenho e escalabilidade;
- Mecanismo flexível de consulta de dados;
- Sem necessidade de gerenciamento de servidores ou aplicações de banco de dados.

Para você ter uma ideia melhor de como funciona o Google Cloud Datastore, segue uma tabela comparativa com o mundo dos bancos de dados relacionais, para que você possa se acostumar com os termos usados nele. Essa comparação foi retirada do site de documentação do Google Cloud Datastore (https://cloud.google.com/datastore/docs/concepts/overview):

Conceito	Datastore	Banco de dados relacional
Categoria do objeto	Tipo (*Kind*)	Tabela (*Table*)
Um objeto	Entitidade (*Entity*)	Linha (*Row*)
Dado individual de um objeto	Propriedade (*Property*)	Campo (*Field*)
Identificação única de um objeto	Chave (*Key*)	Chave primária (*Primary Key*)

É importante aprender esses conceitos do Google Cloud Datastore (*Kind*, *Entity*, *Property* e *Key*), pois quando você for escrever o código em Java para salvar ou pesquisar um objeto, você precisará saber deles para fazer as devidas associações. Nas seções a seguir, você verá como cada um deles funciona na prática.

6.2 CONFIGURANDO O PROJETO PARA TRABALHAR COM DATASTORE

Para demonstração e exemplificação das técnicas de armazenamento e consulta de dados com o Google Cloud Datastore, será utilizado o mesmo projeto `exemplo1` , e o mesmo

serviço de gerenciamento de produtos, porém, todos os métodos serão alterados para buscar os produtos no Google Cloud Datastore. Para isso, execute os passos a seguir:

1. No modelo de produtos, na classe `Product` do pacote `com.siecola.exemplo1.models`, adicione o atributo `id`, como no trecho a seguir, juntamente com seu *getter* e *setter*:

```
private long id;
```

   A unidade fundamental do sistema de armazenamento com o Datastore é uma *Entity* que possui uma identidade imutável, representada por uma chave e por parâmetros que podem ser alterados. Uma entidade pode ser criada, apagada, alterada ou recuperada pelo seu identificador e localizada através de consultas de seus outros parâmetros.

2. Na classe `ProductManager`, crie o método privado `productToEntity` para fazer a conversão de um objeto `Product` para um objeto `Entity`, como no trecho a seguir. Ele será utilizado nos métodos onde for necessário converter um produto para uma entidade `Product` para salvar no Datastore.

```
private void productToEntity (Product product, Entity produ
ctEntity) {
     productEntity.setProperty("ProductID", product.getProdu
ctID());
     productEntity.setProperty("Name", product.getName());
     productEntity.setProperty("Code", product.getCode());
     productEntity.setProperty("Model", product.getModel());
     productEntity.setProperty("Price", product.getPrice());
}
```

3. Crie o método privado `entityToProduct` para fazer a conversão de um objeto `Entity` para um objeto do tipo `Product`. Ele será usado nos métodos onde for necessário buscar uma entidade do Datastore e converter no objeto `Product` a ser retornado nas operações do serviço.

```
 private Product entityToProduct (Entity productEntity) {
     Product product = new Product();
     product.setId(productEntity.getKey().getId());
     product.setProductID((String) productEntity.getProperty(
"ProductID"));
     product.setName((String) productEntity.getProperty("Nam
e"));
     product.setCode(Integer.parseInt(productEntity.getPrope
rty("Code")
             .toString()));
     product.setModel((String) productEntity.getProperty("Mo
del"));
     product.setPrice(Float.parseFloat(productEntity.getProp
erty("Price")
             .toString()));
     return product;
 }
```

1. Ainda na classe `ProductManager`, altere o método `saveProduct`, que é quem implementa a operação para inserir um novo produto. Dessa vez, o produto que chegar como parâmetro no corpo da requisição `HTTP POST` será salvo no Google Cloud Datastore com a execução do código a seguir:

```
 @POST
 @Produces(MediaType.APPLICATION_JSON)
 @Consumes(MediaType.APPLICATION_JSON)
 public Product saveProduct(Product product) {

     DatastoreService datastore = DatastoreServiceFactory
             .getDatastoreService();

     Key productKey = KeyFactory.createKey("Products", "prod
uctKey");
     Entity productEntity = new Entity("Products", productKe
y);

     productToEntity (product, productEntity);

     datastore.put(productEntity);

     product.setId(productEntity.getKey().getId());
     return product;
 }
```

A primeira linha do método obtém uma instância do serviço Datastore para ser utilizada, nesse caso, para inserção de um novo produto. A segunda linha instancia uma chave que será usada na criação da entidade produto. Essa chave ficará incompleta até que a entidade seja inserida no Datastore, onde será possível obter a identificação única de sua criação, o que é realizado na penúltima linha do método.

A terceira linha cria uma nova entidade do tipo `Products` (nesse caso, uma analogia ao nome da tabela é válida), com a chave criada na linha anterior. A quarta linha preenche as propriedades da entidade `productStore` em um esquema chave-valor (que seriam os nomes dos campos e seus valores), sendo finalizado pela inserção do novo produto com a penúltima linha do método.

Evidentemente, o trecho mostrado é simples o suficiente para apenas mostrar a essência da utilização do serviço Datastore, pois algumas verificações poderiam ser feitas, por exemplo, se algum campo mandatório está presente no objeto produto recebido ou se já há algum cadastrado com o mesmo código. Para resumir e fazer uma relação com a tabela comparativa da seção anterior, tem-se que:

- O **tipo** que está sendo colocado no Datastore é o de nome `Products` ;
- A **entidade** é o produto recebido como parâmetro do método `saveProduct` ;
- As **propriedades** possuem os nomes: `ProductID` , `Name` , `Code` , `Model` , `Price` ;
- O valor da **chave** da entidade salva foi atribuída ao atributo `id` , de `product` .

2. Para listar todos os produtos cadastrados, é necessário criar uma consulta no Datastore para que ele retorne todas as

entidades do tipo `Products` , baseado ou não em uma ordenação por uma de suas propriedades. No exemplo a seguir, isso foi feito com base na propriedade `Code` , ordenando a lista de produtos de forma crescente de acordo com esse valor. O método `getProducts` deve então ficar da seguinte forma:

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public List<Product> getProducts() {

    List<Product> products = new ArrayList<>();
    DatastoreService datastore = DatastoreServiceFactory
            .getDatastoreService();

    Query query;
    query = new Query("Products").addSort("Code", SortDirec
tion.ASCENDING);

    List<Entity> productsEntities = datastore.prepare(query
).asList(
            FetchOptions.Builder.withDefaults());

    for (Entity productEntity : productsEntities) {
        Product product = entityToProduct(productEntity);


        products.add(product);
    }

    return products;
}
```

O que aparece de novo nesse método é justamente a busca de todas as entidades, realizada pelo objeto do tipo `Query` , que usa a propriedade `Code` para organizá-las de forma crescente. Com isso, o método `asList` do objeto `datastore` retorna a lista de todas as entidades (produtos) de nome `Products` .

3. Para recuperar apenas um produto do `Datastore` usando seu código como parâmetro de pesquisa, basta alterar o

método getProduct como no trecho a seguir:

```
@GET
@Produces(MediaType.APPLICATION_JSON)
@Path("/{code}")
public Product getProduct(@PathParam("code") int code) {

     DatastoreService datastore = DatastoreServiceFactory
             .getDatastoreService();

     Filter codeFilter = new FilterPredicate("Code", FilterO
perator.EQUAL, code);

     Query query = new Query("Products").setFilter(codeFilte
r);
     Entity productEntity = datastore.prepare(query).asSingl
eEntity();

     if (productEntity != null) {
         Product product = entityToProduct(productEntity);

         return product;
     } else {
         throw new WebApplicationException(Status.NOT_FOUND)
;
     }
}
```

Para localizar uma entidade específica a partir de uma de suas propriedades, é necessário utilizar um filtro. Nesse exemplo, é criado um filtro para a propriedade `Code` , para que seja igual ao código passado como parâmetro pela URL. Após isso, a busca é realizada, requisitando somente uma entidade (produto). Caso ela seja encontrada, o objeto `Product` é preenchido com as informações e retornado como resposta, porém, caso não seja, uma exceção é lançada para indicar que ele não foi encontrado com o código fornecido, por meio do código `HTTP 404 - Not Found` .

4. Para alterar uma entidade específica, deve-se alterar o método `alterProduct` , como no trecho a seguir. Ele é semelhante ao método para buscar somente uma entidade, em termos da

consulta para localizar tal produto a partir do código. A partir daí, as alterações são aplicadas ao produto e este salvo novamente no Datastore:

```
@PUT
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
@Path("/{code}")
public Product alterProduct(@PathParam("code") int code, Pr
oduct product) {
     DatastoreService datastore = DatastoreServiceFactory
             .getDatastoreService();

     Filter codeFilter = new FilterPredicate("Code",
             FilterOperator.EQUAL, code);

     Query query = new Query("Products").setFilter(codeFilte
r);

     Entity productEntity = datastore.prepare(query).asSingl
eEntity();

     if (productEntity != null) {
         productToEntity (product, productEntity);

         datastore.put(productEntity);

         product.setId(productEntity.getKey().getId());
         return product;
     } else {
         throw new WebApplicationException(Status.NOT_FOUND)
;
     }
}
```

Repare na linha `datastore.put(productEntity);` , que salva o produto no Datastore novamente, com as alterações recebidas do modelo recebido no corpo da requisição `HTTP PUT` .

5. Por último, para apagar uma entidade, basta modificar o método `deleteProduct` , como mostra o trecho a seguir:

```
@DELETE
@Produces(MediaType.APPLICATION_JSON)
```

```
 @Path("/{code}")
 public Product deleteProduct(@PathParam("code") int code) {

     DatastoreService datastore = DatastoreServiceFactory
             .getDatastoreService();

     Filter codeFilter = new FilterPredicate("Code",
             FilterOperator.EQUAL, code);

     Query query = new Query("Products").setFilter(codeFilte
r);

     Entity productEntity = datastore.prepare(query).asSingl
eEntity();

     if (productEntity != null) {
         datastore.delete(productEntity.getKey());

         Product product = entityToProduct(productEntity);

         return product;
     } else {
         throw new WebApplicationException(Status.NOT_FOUND)
;
     }
 }
```

Repare que o produto é localizado da mesma forma como foi visto nos métodos para alteração e busca por código. A diferença está em que, caso o produto seja encontrado, o comando `datastore.delete(productEntity.getKey());` é executado para excluí-lo do Datastore. Caso contrário, a mensagem `HTTP 404 Not Found` é enviada.

6.3 TESTANDO NA MÁQUINA LOCAL

Para testar as alterações na sua máquina local de desenvolvimento utilizando o REST Console, basta seguir os mesmos passos mostrados no capítulo anterior, acessando os serviços pelos métodos e URLs mostrados a seguir:

- GET `/api/product/{code}` : para recuperar um

produto de código `{code}` ;

- GET `/api/product` : para recuperar a lista de todos os produtos;
- POST `/api/product` : para inserir um novo produto;
- PUT `/api/product/{code}` : para alterar um produto existente, passando seu código na URL e as alterações no corpo da requisição;
- DELETE `/api/product/{code}` : para apagar o produto com o código passado na URL.

A única diferença é que agora o modelo do produto, que será retornado pelas operações, possui o atributo `id` , ficando como mostra o JSON a seguir. Porém, não é necessário adicionar esse atributo nas operações para inserção e alteração de produtos, pois ele será gerado pelo Datastore quando o produto for inserido.

```
{
    "id": 5629499534213120,
    "productID": "1",
    "name": "Nome 1",
    "model": "Model 1",
    "code": 1,
    "price": 10.0
}
```

Rodando a aplicação no GAE

Publique a aplicação no GAE e faça alguns testes com o serviço de produtos, inserindo e pesquisando.

Método DELETE no GAE

Não faça requisições com o método `DELETE` no GAE com algo configurado no corpo da mensagem. Isso vai fazê-lo dar exceção, pois esse método, no `exemplo1` não foi configurado para receber uma requisição com uma mensagem no corpo. O Jersey tem esse comportamento, pois, pela referência do REST, a operação DELETE não deve ter uma mensagem no corpo da requisição.

6.4 ÍNDICES DO DATASTORE

O Google Cloud Datastore trabalha com índices para cada consulta que é feita pela aplicação. Para tornar as buscas mais rápidas, você pode especificar esses índices através de um arquivo de configuração, chamado `datastore-indexes.xml` a ser criado na pasta `WEB-INF`. Esses índices devem ser criados contendo 3 características principais:

- Tipo da entidade a ser indexada;
- Propriedade da entidade usada na busca;
- Ordenação das entidades retornadas na busca.

No exemplo mostrado, a aplicação realiza consultas de entidades do tipo `Product`, baseadas na propriedade `Code` e ordenada de forma crescente, no caso do método `getProducts`. Para esse tipo de busca, que utiliza apenas uma propriedade como base de pesquisa, não é necessário criar índices, mas por exemplo, caso a busca também necessitasse de uma ordenação pela propriedade `Model`, isso seria necessário.

Dessa forma, para tornar essas buscas com mais de uma

propriedade mais rápidas, basta criar o arquivo de configuração de índices, conforme os passos a seguir:

1. Crie o arquivo `datastore-indexes.xml` na pasta `WEB-INF`;
2. Crie o esqueleto desse arquivo, como mostra o trecho a seguir:

```
<?xml version="1.0" encoding="utf-8"?>
<datastore-indexes
  autoGenerate="true">

</datastore-indexes>
```

3. Dentro da tag `datastore-indexes`, crie o índice para a consulta realizada:

```
<datastore-index kind="Product" ancestor="false">
    <property name="Code" direction="asc" />
    <property name="Code" direction="asc" />
</datastore-index>
```

   Sendo assim, cada elemento `datastore-index` representa um índice que auxilia o Datastore na busca para esse tipo de entidade através das propriedades informadas.

4. Publique a aplicação novamente no Google App Engine, para que você possa visualizar as entidades e esse índice que foi criado.

Exercício proposto

Altere o serviço de gerenciamento de produtos do projeto `exemplo1` para que realize as seguintes verificações:

- Na operação de alterar o produto, exigir que o atributo `id` esteja presente e preenchido com um valor válido. Caso ele não tenha sido enviado, responder com `HTTP 400 Bad Request`.
- Verificar antes de cadastrar ou alterar um produto, se já

não há outro produto cadastrado com o mesmo código. Caso já exista algum produto com o mesmo código, as operações deverão responder com o código `HTTP 400 Bad Request`.

A resolução desse exercício está no repositório do livro, no endereço:

https://github.com/siecola/GAEBook

6.5 ADMINISTRANDO O DATASTORE NO GAE

O console do Google App Engine possui uma parte dedicada para administração do Datastore. Ele pode ser acessado clicando-se no menu suspenso no canto superior esquerdo da tela, na seção `Datastore` da divisão `Datastore`.

Essa seção do console do GAE fornece ferramentas como:

Dashboard

Essa é a primeira opção do console de administração do Google Cloud Datastore. Nela você pode obter estatísticas sobre os acessos aos tipos e as entidades. Essas estatísticas são geradas a cada 24 horas.

Gerenciamento e pesquisa de entidades

Essa ferramenta pode ser acessada na opção `Entities`. Com ela, é possível:

- Visualizar os tipos existentes;
- Listar as entidades cadastradas de cada tipo, como mostra a figura a seguir:

Figura 6.1: Listando entidades no Datastore

- Filtrar as entidades a partir de qualquer de suas propriedades:

Figura 6.2: Filtrando entidades no Datastore

- Apagar uma entidade, selecionando-a e clicando no botão `Delete` ;
- Alterar uma entidade, clicando-se no seu ID:

Figura 6.3: Editando uma entidade no Datastore

- Realizar buscas usando o `Google Query Language` ou GQL. Para uma ajuda sobre o GQL, consulte https://cloud.google.com/datastore/docs/apis/gql/gql_reference.

Figura 6.4: Filtrando entidades no Datastore com GQL

- Criar novas entidades:

Figura 6.5: Criando entidades no Datastore com GQL

Visualização dos índices de pesquisa

Nesta seção, é possível visualizar os índices criados para cada entidade:

Indexes

Below are the composite indexes for this application. These indexes are managed in your app's index configuration file. Learn more

Kind	Indexes	Size	Entries	Status
Product	Code ▲ + Model ▲	–	–	Serving

Figura 6.6: Visualizando os índices

Administração

Nesta seção, você pode fazer backup, restaurar, copiar e apagar entidades de forma massiva e ainda desabilitar as operações de escrita no Datastore na instância onde você está administrando.

VERSÃO SIMPLIFICADA DO CONSOLE LOCAL DO DATASTORE

Há uma versão simplificada do console de administração do Datastore que pode ser acessada na sua máquina de desenvolvimento, quando a aplicação está em execução. Ele pode ser acessado pelo endereço: http://localhost:8888/_ah/admin/datastore

6.6 CONCLUSÃO

Neste capítulo, você aprendeu uma importante tarefa: como persistir dados no Google Cloud Datastore. Além disso, você também aprendeu técnicas de como pesquisar entidades salvas no Datastore, validações e como administrar tudo por meio do console de administração do GAE.

No próximo capítulo, você verá como gerar mensagens de log através de uma aplicação no GAE, e como visualizar essas mensagens no console de administração do GAE.

CAPÍTULO 7

GERANDO MENSAGENS DE LOG

Por muitas vezes, é necessário depurar ou registrar eventos em aplicações que estão sendo executadas no GAE, para que possam ser analisados em um momento futuro. Em algumas situações, apenas as estatísticas, gráficos de requisições, trace e logs gerados e exibidos pelo console do GAE (que são muito bons) não são suficientes para descobrir a causa raiz de um problema.

Para isso, é necessário lançar mão da geração de mensagens log pela aplicação. Essas mensagens então poderão ser visualizadas no console do GAE. Para geração de logs pela aplicação, pode se usar o `java.util.logging.Logger` , como será mostrado neste capítulo.

7.1 CONFIGURANDO O PROJETO PARA GERAÇÃO LOGS

Neste capítulo, você gerará mensagens de log nas operações do serviço de gerenciamento de produtos para serem visualizadas no console do GAE. Mas antes, é necessário realizar algumas configurações no projeto `exemplo1` :

Abra o arquivo `src/main/webapp/WEB-INF/logging.properties` , que é onde são feitas as configurações para a geração de log, e veja que há a propriedade `.level` , configurada inicialmente com o valor padrão `WARNING` .

```
.level = WARNING
```

Há 5 níveis de severidade de log que podem ser configurados nessa propriedade. São eles:

- `DEBUG`
- `INFO`
- `WARNING`
- `ERROR`
- `CRITICAL`

O nível `CRITICAL` é o mais severo e o `DEBUG` o de menor severidade.

Isso significa que é possível gerar mensagens com esses níveis, para depois serem filtrados na hora de serem efetivamente gravados no mecanismo de armazenamento dessas mensagens de logs.

Nesses 5 níveis de log, há uma ordem na filtragem para que as mensagens apareçam. Por exemplo, configurando o nível de log como `WARNING`, as mensagens geradas como `INFO` e `DEBUG` não aparecerão, mas sim as geradas com `WARNING`, `ERROR` e `CRITICAL`. Na interface de monitoramento de logs do console do GAE, é possível selecionar as mensagens pelo nível de severidade delas, como será visto mais adiante ainda neste capítulo.

Como em qualquer outra aplicação, é muito importante saber definir o nível de severidade na geração das mensagens de log, bem como na configuração de filtragem, pois em um primeiro momento, muitas mensagens sendo geradas podem afetar o desempenho geral da aplicação.

No GAE, ainda há dois fatores que devem ser levados em consideração, pois há limites na utilização da plataforma se você estiver usando-a no modo gratuito, como:

- **Tempo de armazenamento das mensagens**: as mensagens de log ficam armazenadas por até 90 dias na plataforma. Caso sua aplicação esteja habilitada para cobranças (ou seja, não está no modo gratuito), essas mensagens poderão ficar armazenadas por até 365 dias.
- **Tamanho máximo do armazenamento das mensagens**: no modo gratuito, há um limite máximo de 1 GB de armazenamento das mensagens de log. Caso chegue ao limite, as primeiras mensagens vão sendo apagadas à medida que novas mensagens aparecem, em um esquema de reciclagem das mais antigas.

Também é possível, nesse arquivo, definir o nível de log específico para classes pertencentes a um pacote, por exemplo:

```
com.siecola.exemplo1.services.level = DEBUG
```

Adicione a configuração a seguir no arquivo `src/main/webapp/WEB-INF/appengine-web.xml`, responsável por indicar ao mecanismo de log sobre qual arquivo de configuração deve ser utilizado:

```
<system-properties>
    <property name="java.util.logging.config.file" value="WEB-INF/
logging.properties"/>
</system-properties>
```

Repare que o arquivo é aquele citado no item anterior, onde pode ser feita a configuração do nível das mensagens de log a serem geradas.

7.2 MÉTODOS PARA GERAÇÃO DE LOGS

Agora que você já entendeu um pouco como funciona o mecanismo de log do GAE, é hora de entender como fazer para gerar tais mensagens no código. Como exemplo, será utilizado o

serviço de gerenciamento de produtos, onde serão colocados comandos dentro dos métodos da classe `ProductManager` .

Dessa forma, toda vez que uma operação desse serviço for acessada, a aplicação vai gerar uma mensagem de log, que poderá ser visualizada no console do GAE, como será mostrado na próxima seção deste capítulo.

A primeira coisa a fazer é importar a biblioteca `java.util.logging.Logger` na classe `ProductManager` :

```
import java.util.logging.Logger;
```

Para utilizar os métodos de geração de log da classe *Logger*, crie um atributo privado e estático dentro da classe `ProductManager` :

```
private static final Logger log = Logger.getLogger("ProductManager'
);
```

A chamada `Logger.getLogger("ProductManager")` cria ou localiza uma instância do mecanismo de log com o nome fornecido; nesse caso foi utilizado o próprio nome da classe.

Pronto! Agora é só usar o atributo `log` com um dos seguintes métodos:

```
log.finest("Mensagem de nível DEBUG");
log.finer("Mensagem de nível DEBUG");
log.fine("Mensagem de nível DEBUG");
log.config("Mensagem de nível DEBUG");
log.info("Mensagem de nível INFO");
log.warning("Mensagem de nível WARNING");
log.severe("Mensagem de nível ERROR");
```

Veja que as mensagens de nível de `DEBUG` podem ser geradas pelos métodos `finest` , `finer` , `fine` ou `config` . Para a mensagem de nível `ERROR` , deve ser utilizado o método `severe` . Já as mensagens de nível `INFO` e `WARNING` possuem os seus métodos próprios.

Ainda, ao serem utilizados os métodos de escrita na saída padrão, a mensagem será encarada como do nível `INFO` :

```
System.out.println("Mensagem INFO");
```

E ao ser usado o métodos de escrita na saída de erro, a mensagem será encarada como do nível `WARNING` :

```
System.err.println("Mensagem WARNING");
```

As mensagens do nível `CRITICAL` são reservadas para exceções que não forem tratadas, ou seja, quando isso acontece, o mecanismo de log do GAE encarará essas mensagens com o nível `CRITICAL` .

Vale lembrar de que as mensagens que efetivamente serão gravadas no mecanismo de log serão aquelas que estiverem com um nível de log acima do filtro configurado na propriedade `.level` do arquivo `src/main/webapp/WEB-INF/logging.properties` .

Veja a seguir como pode ficar o método `deleteProduct` , recheado de mensagens de log para registrar tudo o que pode acontecer quando ele é chamado, com mensagens de níveis diferentes, dependendo da gravidade ou razão do evento.

```
    @DELETE
    @Produces(MediaType.APPLICATION_JSON)
    @Path("/{code}")
    public Product deleteProduct(@PathParam("code") int code) {

        //Mensagem 1 - DEBUG
        log.fine("Tentando apagar produto com código=[" + code + "
]");

        DatastoreService datastore = DatastoreServiceFactory
                .getDatastoreService();

        Filter codeFilter = new FilterPredicate("Code",
                FilterOperator.EQUAL, code);

        Query query = new Query("Products").setFilter(codeFilter);

        Entity productEntity = datastore.prepare(query).asSingleEn
```

```
tity();

        if (productEntity != null) {
            datastore.delete(productEntity.getKey());

            //Mensagem 2 - INFO
            log.info("Produto com código=[" + code + "] apagado co
m sucesso");

            Product product = entityToProduct(productEntity);

            return product;
        } else {
            //Mensagem 3 - ERROR
            log.severe ("Erro ao apagar produto com código=[" + co
de + "]. Produto não encontrado!");

            throw new WebApplicationException(Status.NOT_FOUND);

        }
    }
```

Veja que a primeira mensagem de log (`Mensagem 1 - DEBUG`), no início do método, foi gerada com o método `fine` , o que significa que ela terá o nível `DEBUG` no GAE. A mensagem que diz que o produto foi apagado com sucesso (`Mensagem 2 - INFO`) foi gerada com o nível `INFO` . Já a última mensagem (`Mensagem 3 - ERROR`), antes do lançamento da exceção lançada caso o produto não seja encontrado, foi construída com o método `severe` , o que gera uma mensagem com o nível `ERROR` .

No caso desse método, haveria 3 possibilidades de configuração do filtro de severidade das mensagens. Seriam eles:

- `FINE` : todas as 3 mensagens apareceriam, pois é o nível mais baixo, correspondente ao `DEBUG` , o que faz com que todas as mensagens com nível acima desta sejam gravadas no log. Isso faz com que muitas mensagens apareçam, por isso é interessante configurar esse nível apenas para o pacote que você está depurando;

- `INFO` : somente as mensagens 2 e 3 seriam gravadas no log;
- `ERROR` : somente a mensagem 3 seria gravada no log, pois as outras duas (`INFO` e `DEBUG`) possuem níveis de severidade menor do que `ERROR` .

> **ESCOLHA DO NÍVEL DE SEVERIDADE DAS MENSAGENS**
>
> A escolha do nível de severidade das mensagens não é totalmente objetiva e pode variar de programador para programador, mas é importante ter bom senso na hora de fazer essa definição, levando em conta o que já foi dito neste capítulo, como quantidade de mensagens e o tamanho que vão ocupar em disco.

Exercício proposto

Altere os outros métodos que implementam o serviço de produtos para gerarem mensagens de log, principalmente quando gerarem exceção. Execute a aplicação na sua máquina local de desenvolvimento e veja as mensagens sendo geradas na aba `console` do Eclipse. Altere o nível de severidade da configuração no arquivo `src/main/webapp/WEB-INF/logging.properties` , e veja seus efeitos.

7.3 VISUALIZANDO AS MENSAGENS DE LOG NO GAE

Nesta seção, você aprenderá como visualizar as mensagens no console do GAE, mas publique a aplicação e faça alguns testes com as operações onde você colocou as mensagens, por exemplo, a

operação de apagar produtos.

No console do GAE, acesse o menu no canto superior direito, clicando na opção `Logging` , que fica no conjunto `Operations` .

Logo no topo da página onde os logs são exibidos, existem algumas opções para filtrar as mensagens que foram geradas pela aplicação. São elas:

- **Versão da aplicação:** aqui você pode selecionar, dentre as opções de aplicações do Google App Engine, qual versão da aplicação você quer exibir os logs, como mostra a figura a seguir:

Figura 7.1: Mensagens de log por versão da aplicação

Repare que, como foram publicadas apenas duas versões da aplicação (versão 1 e versão 1-1), somente elas aparecem como opção. Aqui você também pode optar por ver todas as versões, selecionando a opção `All versions`

- **Tipo de log:** basicamente eles se dividem em log de atividade, como publicação de novas versões ou alterações, e logs de requisições, que são os mais comuns, dado que uma aplicação no GAE basicamente responde a requisições HTTP.

- **Nível de severidade da mensagem de log:** esse é um filtro importante, quando você possui muitas mensagens de log e deseja procurar apenas pelas mensagens com o nível `INFO` , por exemplo, porque sabe que é nesse nível que está a informação que você procura.

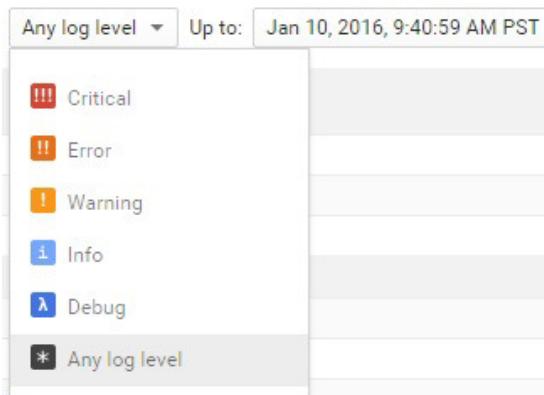



Figura 7.2: Nível de log

- **Período de geração da mensagem de log:** você pode escolher a data/hora da mensagem de log que deseja exibir, muito útil para analisar eventos que aconteceram com dias e/ou horários bem conhecidos.

- **Campo editável:** você pode digitar um texto para filtrar a mensagem de log, como no exemplo a seguir, onde as mensagens são filtradas pela URL da requisição que as

geraram:

Figura 7.3: Filtrando log pela URL da requisição

Repare que somente as mensagens de log que foram geradas pela requisição à URL `api/products/1` é que apareceram como resultado do filtro. Isso é muito útil quando você deseja ver o que aconteceu quando um produto `X` foi acessado, por qualquer método.

Você também pode combinar filtros, como no exemplo a seguir, onde só aparecem mensagens de log de requisições `GET` na URL `api/products`:

Figura 7.4: Combinando filtros de log

É possível expandir cada mensagem da lista para ver detalhes do que acontece. O ícone localizado à esquerda da mensagem refere-se à sua severidade, como nas opções de configuração do filtro para essa opção. Nos detalhes da mensagem expandida, é possível ver qual foi o método HTTP usado para a requisição, a URL de acesso, detalhes do cliente, o código HTTP de retorno, tempo de resposta, consumo de CPU e algumas outras coisas mais.

Veja um exemplo onde foi gerada também uma mensagem pelo método `deleteProduct` , informando que o produto de código 1 foi apagado com sucesso:

Figura 7.5: Mensagem no log

Há um link na linha da mensagem de log, logo à esquerda da URL da requisição. Se você clicar nele, será redirecionado para a página de `Trace` , onde poderá analisar com muito mais clareza tudo o que aconteceu, como pode ser visto na figura a seguir:

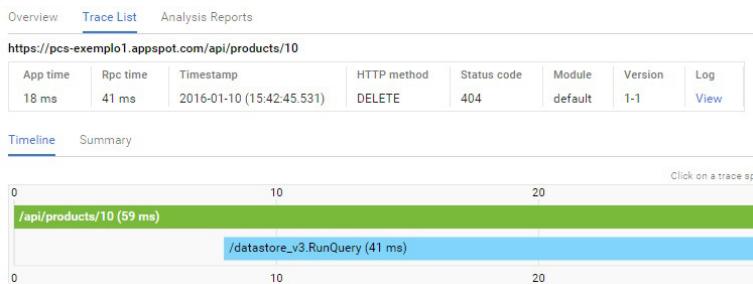



Figura 7.6: Trace da requisição

Repare nessa página que é possível ter um gráfico com o tempo que cada operação levou para executar, desde o tratamento da requisição em si pelo método, como pelo tempo em que houve a

consulta ao serviço de Datastore.

Saber utilizar bem as ferramentas de logs e trace pode economizar muito tempo na depuração de um erro, ou no processo de otimização de um tempo de resposta muito longo de uma requisição. **Não deixe de explorar essas funcionalidades que o GAE oferece!**

7.4 CONCLUSÃO

Neste capítulo, você aprendeu a gerar mensagens de log e visualizá-las no console do GAE, realizando filtros das mais diversas formas. Você viu como o GAE é detalhando nas informações sobre o que aconteceu em cada requisição à aplicação. Tudo para que torne o trabalho do desenvolvedor ou do administrador da aplicação mais fácil para descobrir um erro, ou otimizar o tempo de resposta da aplicação. Saber usar as ferramentas de análise de logs oferecidas pelo GAE é um grande diferencial de um profissional que deseja trabalhar com essa plataforma.

No próximo capítulo, você aprenderá como proteger o acesso aos serviços da aplicação `exemplo1`, utilizando um mecanismo de autenticação simples, o `HTTP Basic Auth`.

Capítulo 8

PROTEGENDO SERVIÇOS COM HTTP BASIC AUTHENTICATION

O serviço de gerenciamento de produtos, que foi criado e publicado no GAE, está acessível a qualquer usuário ou sistema, sem nenhum mecanismo de autenticação para evitar acessos não autorizados. Aliás, não há nenhum tipo de solicitação de autorização. Isso significa que, caso você estivesse construindo um sistema proprietário, exclusivo para uso de um cliente, qualquer pessoa que tivesse conhecimento dele, e que estivesse conectada à internet, poderia utilizá-lo. E se você ou o seu cliente estivessem pagando pela infraestrutura do GAE, poderia haver custos por excesso de requisições de usuários e locais desconhecidos.

Esse fato não seria um problema se você estivesse disponibilizando uma API pública, ou seja, algo que fosse servir para qualquer sistema consumir de forma gratuita. Aí você pode pensar: "Mas assim eu teria de pagar para ter essa API publicada no GAE!?". Não, se você não habilitar a cobrança na sua aplicação. O que pode acontecer é a sua aplicação chegar a um determinado limite diário de consumo do GAE, como número máximo de requisições ou tempo de CPU, e ficar inacessível até o próximo período de contabilização (diário ou mensal).

Ainda bem que, por padrão, as aplicações criadas no GAE não

ficam com os mecanismos de cobrança habilitados, então você pode ficar despreocupado.

Mas se você vai desenvolver uma API e, por qualquer que seja o motivo, não deseja liberar o acesso a qualquer sistema, é possível implementar um mecanismo de autenticação e autorização. Há várias questões que devem ser levadas em consideração na escolha de tal mecanismo, como:

- Nível de segurança desejado, pois obviamente existem mecanismos com diferentes técnicas que permite uma maior ou menor segurança;
- Compatibilidade com as aplicações clientes que vão consumir o serviço;
- Carga do servidor no processo de tratamento e autenticação das requisições;
- Quantidade de requisições a serem feitas pelo cliente para poder ser autenticado. O mecanismo HTTP Digest Authentication com MD5, por exemplo, oferece um nível de segurança consideravelmente elevado, mas requer que toda requisição seja negada pelo servidor na primeira vez, que devolve uma chave ou semente aleatória, e assim o cliente a refaz com o usuário e senha gerados a partir dessa semente.

No capítulo *Protegendo serviços com OAuth 2*, será mostrado um mecanismo onde o cliente adquire um `token`, utilizando suas credenciais de acesso. O cliente então usa esse token, que é válido apenas por um tempo determinado, para autenticar suas requisições no servidor. Isso será feito com ajuda da plataforma do Google App Engine.

Por enquanto, você conhecerá, neste capítulo, sobre o mecanismo **HTTP Basic Authentication**, para iniciar a questão de

autenticação e autorização de acesso de usuários a serviços REST com o Jersey. Dessa forma, também será mostrado como construir um mecanismo usando filtros, que serão chamados em cada requisição HTTP. Dentro desses filtros é que será implementada a verificação das credenciais de acesso ao serviço.

8.1 O QUE É HTTP BASIC AUTHENTICATION

O mecanismo HTTP Basic Authentication é muito simples de ser implementado, tanto do lado do servidor como do cliente. Ele apenas requer que, em toda requisição HTTP, o cliente envie o cabeçalho `Authorization` , como mostrado a seguir:

```
Authorization: Basic QWRtaW46QWRtaW4=
```

O campo `Authorization` deve conter o esquema de autenticação, `Basic` , como diz o próprio nome do esquema, e as credenciais de acesso (usuário e senha), codificados em Base64, no formato "usuario:senha".

O servidor para tratar a requisição e autorizá-la ou não, deve executar os seguintes passos:

1. Verificar a existência do cabeçalho com o nome `Authorization` ;
2. Ler e validar o esquema de autenticação, que deve ser do tipo `Basic` ;
3. Ler a string com as credenciais em formato Base64;
4. Decodificar as credenciais em formato Base64 e obter o usuário e a senha enviados pelo cliente;
5. De posse do usuário e da senha, o servidor pode buscar em uma base de dados, para verificar se as credenciais estão corretas e o usuário tem permissão de acesso, baseado em regras ou papéis.

Como você pode ver, esse mecanismo é bem simples, com apenas alguns passos a serem executados por parte do servidor, o que o torna muito leve, mas com algumas desvantagens:

- Toda requisição possui o mesmo valor no cabeçalho `Authorization` , com os valores do usuário e da senha codificados sempre da mesma forma;
- O usuário e a senha não estão criptografados, o que permite que eles sejam facilmente decodificados aos valores originais;
- Se alguma requisição do cliente for interceptada, por exemplo, com um *sniffer* de rede, as credenciais de acesso estarão expostas e poderão ser decodificadas com quase nenhum esforço.

Se você possui uma preocupação com o fato de as requisições dos clientes da sua API poderem ser interceptadas, é melhor utilizar um mecanismo de autenticação mais forte, como OAuth ou HTTP Digest. Há ainda quem recomende usá-lo em conjunto com algum mecanismo de criptografia da requisição como um todo, como o SSL. Entretanto, a análise sobre o que é mais ou menos custoso para o servidor é um assunto mais longo, e não está no escopo do livro.

> Se o rigor da segurança não for uma preocupação grande, o HTTP Basic Authentication é um bom mecanismo de proteção de serviços REST, que aliás, é muito utilizado em aplicações com requisitos de baixa segurança e menor consumo de recursos do servidor.

8.2 CONFIGURANDO O PROJETO PARA HTTP BASIC AUTHENTICATION

Agora que você já sabe o que é e como funciona o mecanismo HTTP Basic Authentication, é hora utilizá-lo no projeto para proteger o serviço de gerenciamento de produtos. Primeiramente com o básico, que é simplesmente configurar o projeto para proteger todas as requisições a esse serviço, verificando um usuário e sua senha, fixos no código, para que os conceitos primordiais possam ser entendidos. Para isso, siga os passos a seguir:

1. Abra o arquivo `src/main/webapp/WEB-INF/web.xml` e localize a declaração do servlet de nome `Exemplo1Services` :

```
<servlet>
    <servlet-name>Exemplo1Services</servlet-name>
```

2. No final da tag `servlet` , antes do parâmetro `<load-on-startup>1</load-on-startup>` , adicione o trecho a seguir para dizer ao Jersey que todas as requisições ao servlet `Exemplo1Services` , que na verdade intercepta tudo o que chega na URL `/api` , sejam antes tratadas pelo filtro presente em `com.siecola.exemplo1.authentication.AuthFilter` :

```
<init-param>
    <param-name>jersey.config.server.provider.classnames</p
aram-name>
    <param-value>com.siecola.exemplo1.authentication.AuthFi
lter;org.glassfish.jersey.filter.LoggingFilter</param-value>
</init-param>
```

   A classe `AuthFilter` será criada no pacote `com.siecola.exemplo1.authentication` do projeto, e será responsável por realizar o processo de autenticação de todas as requisições feitas na URL `/api/*` . Essa classe será implementada na próxima seção deste capítulo.

O arquivo `src/main/webapp/WEB-INF/web.xml` , nesse momento, deverá ficar assim:

```
<servlet>
    <servlet-name>Exemplo1</servlet-name>
```

```
        <servlet-class>com.siecola.exemplo1.Exemplo1Servlet</servl
et-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>Exemplo1</servlet-name>
        <url-pattern>/exemplo1</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
    </welcome-file-list>

    <servlet>
        <servlet-name>Exemplo1Services</servlet-name>
        <servlet-class>org.glassfish.jersey.servlet.ServletContain
er</servlet-class>
        <init-param>
            <param-name>jersey.config.server.provider.packages</pa
ram-name>
            <param-value>com.siecola.exemplo1.services</param-value
>
        </init-param>
        <init-param>
            <param-name>org.glassfish.jersey.api.json.POJOMappingF
eature</param-name>
            <param-value>true</param-value>
        </init-param>
        <init-param>
            <param-name>jersey.config.server.provider.classnames</
param-name>
            <param-value>com.siecola.exemplo1.authentication.AuthF
ilter;org.glassfish.jersey.filter.LoggingFilter</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Exemplo1Services</servlet-name>
        <url-pattern>/api/*</url-pattern>
    </servlet-mapping>
```

É importante dizer que há, pelo menos, mais duas outras formas de se construir uma aplicação com HTTP Basic Authentication, utilizando o Jersey. Essa é uma forma amigável e que se adapta bem com a plataforma do App Engine.

8.3 CRIANDO A CLASSE DE FILTRO E PROTEGENDO OS SERVIÇOS

Como mostrado na seção anterior, o aplicação foi configurada no arquivo `src/main/webapp/WEB-INF/web.xml` , para que todas as requisições na URL `api/*` sejam antes tratadas pelo filtro presente na classe `AuthFilter` . Então, agora é preciso criá-lo com seu esqueleto básico. Para isso, siga os passos:

1. Crie o pacote `com.siecola.exemplo1.authentication` no projeto.

2. Crie a classe `AuthFilter` dentro desse pacote:

   ```
   import javax.ws.rs.container.ContainerRequestFilter;

   public class AuthFilter implements ContainerRequestFilter {

       private static final String ACCESS_UNAUTHORIZED = "Você
   não tem permissão para acessar esse recurso";

   }
   ```

   A classe deve implementar a interface `javax.ws.rs.container.ContainerRequestFilter` , que possui o método `filter` .

3. Crie o método `filter` , que será invocado, segundo as configurações feitas no arquivo `src/main/webapp/WEB-INF/web.xml` , quando uma requisição chegar à URL `api/*` :

   ```
   @Override
   public void filter(ContainerRequestContext requestContext)
   {

   }
   ```

   Esse método recebe como parâmetro um objeto do tipo `ContainerRequestContext` , que nesse momento será usado apenas para dizer ao Jersey se a requisição foi aceita ou não.

4. Crie o método privado `decode` , que será utilizado apenas para decodificar e separar o usuário e a senha da requisição do formato Base64:

```
private String[] decode(String auth) {
    auth = auth.replaceFirst("[B|b]asic ", "");

    byte[] decodedBytes = DatatypeConverter.parseBase64Bina
ry(auth);

    if (decodedBytes == null || decodedBytes.length == 0) {
        return null;
    }

    return new String(decodedBytes).split(":", 2);
}
```

Repare que ele recebe uma string, que deve conter o conteúdo do cabeçalho `Authorization` da requisição. Como foi dito anteriormente, esse conteúdo estará no seguinte formato:

```
Authorization: Basic QWRtaW46QWRtaW4=
```

O método `decode` , então, extrai a string depois do valor `Basic` , chamando o método `DatatypeConverter.parseBase64Binary(String auth)` , que decodifica o formato Base64 e devolve um array de bytes.

Se ele não for vazio, o método `decode` devolve um array de String de duas posições, sendo a primeira com o usuário e a segunda com a senha.

5. Nesse momento, o método `filter` terá apenas uma verificação se o cabeçalho `Authorization` está presente ou não, e se o usuário e a senha conferem com um valor fixo colocado no código. Isso será feito para simplificar o entendimento, a análise do mecanismo HTTP Basic Authentication e de outros conceitos que você ainda vai ver neste capítulo. Veja como ele deve ficar:

```
 @Override
 public void filter(ContainerRequestContext requestContext)
{
     String auth = requestContext.getHeaderString("Authoriza
tion");

     if (auth == null) {
         requestContext.abortWith(Response.status(Response.S
tatus.UNAUTHORIZED)
                 .entity(ACCESS_UNAUTHORIZED).build());
         return;
     }

     String[] loginPassword = decode(auth);

     if (loginPassword == null || loginPassword.length != 2)
 {
         requestContext.abortWith(Response.status(Response.S
tatus.UNAUTHORIZED)
                 .entity(ACCESS_UNAUTHORIZED).build());
         return;
     }

     if (loginPassword[0].equals("Admin")
             && loginPassword[1].equals("Admin")) {
         return;
     } else {
         requestContext.abortWith(Response.status(Response.S
tatus.UNAUTHORIZED)
                 .entity(ACCESS_UNAUTHORIZED).build());
     }
 }
```

Somente se o usuário e a senha, depois de terem sido decodificados do formato Base64, baterem com o que está fixo no código, é que o método retorna sem nenhuma alteração no contexto da requisição, pela variável `requestContext` do tipo `ContainerRequestContext` . Nos demais casos - como ausência do cabeçalho `Authorization` ; usuário ou senha vazio; usuário ou senha incorretos -, é sinalizado ao contexto da requisição que ela deve ser abortada, por meio do método `abortWith` , que recebe como parâmetro um valor do tipo `Response` .

```
 requestContext.abortWith(Response.status(Response.Status.UN
AUTHORIZED)
     .entity( ACCESS_UNAUTHORIZED).build());
```

Nele, é possível definir o código HTTP de resposta, que nesse caso foi utilizado sabiamente o `HTTP 401 UNAUTHORIZED` , indicando que a requisição não foi autorizada e, por isso, ela **não prosseguirá até o seu destino**, que seria algum método que trata a operação do serviço de gerenciamento de produtos. No objeto `Response` , ainda é possível colocar uma frase, indicando o motivo de a requisição ter sido respondida com esse código HTTP.

8.4 TESTANDO O SERVIÇO DE PRODUTOS COM HTTP BASIC AUTHENTICATION

Agora que os serviços da aplicação `exemplo1` já estão protegidos com o mecanismo de autenticação HTTP Basic Authentication, acesse-os com o REST Console e veja que, sem inserir as credenciais de acesso, toda requisição é negada pela aplicação:

Figura 8.1: Resposta com HTTP 401 UNAUTHORIZED

Repare que o cabeçalho `Status Code` da mensagem de resposta à requisição - que foi um `GET` na URL `api/products` - na tentativa de listar todos produtos, veio com o código `HTTP 401 UNAUTHORIZED`. Esse código de resposta foi gerado pelo método `filter` criado anteriormente, mais especificamente aqui:

```
String auth = requestContext.getHeaderString("Authorization");

if (auth == null) {
    requestContext.abortWith(Response.status(Response.Status.UNAUT
HORIZED)
            .entity(ACCESS_UNAUTHORIZED).build());
    return;
}
```

Como o `if` testa a presença do campo `Authorization` na requisição, e ele não foi colocado, como mostra a figura a seguir, a requisição é então negada com o código `HTTP 401 UNAUTHORIZED`:

```
1. Accept: */*
2. Connection: keep-alive
3. Content-Type: application/xml
4. Origin: chrome-extension: //rest-console-id
5. User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64)
```

Figura 8.2: Requisição sem cabeçalho Authorization

Para fazer com que as requisições possam ser aceitas pela aplicação `exemplo1`, basta inserir as credenciais de acesso na seção `Authorization` do REST Console, como nos passos a seguir:

1. Na seção `Authorization` do REST Console, clique no botão `Basic Auth`;

2. Na janela que aparecer, preencha com as credenciais de acesso aos serviços da aplicação `exemplo1`, que estão fixos como sendo `Admin` para usuário e senha:

Figura 8.3: Colocando credenciais de acesso

3. Clique no botão `Set Header`, para que o REST Console crie o campo `Authorization`, com as credenciais de acesso codificadas no formato Base64. Dessa forma, o REST Console ficará configurado como mostra a figura seguir:

Figura 8.4: Cabeçalho Authorization

4. Agora que o REST console está configurado para enviar o cabeçalho `Authorization` , com as informações necessárias para que o mecanismo `HTTP Basic Authentication` da aplicação `exemplo1` aceite-as, basta acessar qualquer uma de suas operações, como no exemplo adiante, onde é feito um `GET` na URL `api/products` :

```
1. Authorization: Basic QWRtaW46QWRtaW4=
2. Accept: */*
3. Connection: keep-alive
4. Content-Type: application/xml
5. Origin: chrome-extension: //rest-console-id
6. User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64)
```

Figura 8.5: Requisição com cabeçalho Authorization

Dessa forma, no método `filter` da classe `com.siecola.exemplo1.authentication.AuthFilter` , a execução chegará até o fim, onde ele retornará sem abortar a requisição, executando os seguintes passos:

- Teste para ver se o campo `Authorization` existe;

- Decodificação das credenciais do formato Base64;
- Verificação se usuário e senhas não estão vazios;
- Verificação se o usuário e a senha conferem (Admin/Admin), feito pelo trecho a seguir:

```
if (loginPassword[0].equals("Admin")
    && loginPassword[1].equals("Admin")) {
    return;
}
```

Isso fará com que todas as requisições sejam direcionadas para seu destino, ou seja, os métodos das classes configuradas com os respectivos caminhos.

BASE DE DADOS DE USUÁRIOS

É importante ressaltar que, até o momento, a aplicação `exemplo1` não possui base de dados de usuários para focar as explicações e o entendimento dos conceitos na construção dos serviços REST com o Jersey. Por isso, as informações de usuário, senha e papel (que será acrescentado na próxima seção), estão fixos no código.

No capítulo *Adicionando o serviço de usuários*, será construído um serviço para gerenciamento de usuários, e o mecanismo de autenticação será alterado para usar sua base de dados.

8.5 ADICIONANDO ANOTAÇÕES PARA CONTROLE DE PERMISSÕES E PAPÉIS

O serviço de gerenciamento de produtos já está protegido com `HTTP Basic Authentication` , mas todas as operações possuem as mesmas regras de segurança, sem nenhuma diferenciação do

usuário que acessa a operação em si. Imagine que você precisasse implementar as seguintes diretrizes:

- A listagem de todos os produtos ou de somente um, pelo seu código, é permitida a qualquer usuário, mesmo que não tenha se autenticado;
- O cadastramento e alteração de um produto só pode ser feita por usuários autenticados com o papel `USER` ;
- A exclusão de um produto só pode ser feita por um usuário com o papel `ADMIN`

Isso tudo pode ser feito com algumas alterações no método `filter` da classe `com.siecola.exemplo1.authentication.AuthFilter` , e com anotações nos métodos que implementam essas operações na classe `com.siecola.exemplo1.services.ProductManager` .

Utilizar papéis para os usuários é uma forma de diferenciá-los, dando a eles permissões ou restrições, seja para todas ou algumas operações de um serviço REST.

Relembrando e associando conceitos de implementações REST

Apenas para facilitar o entendimento sobre os próximos passos que serão apresentados neste capítulo, lembre-se de que:

- No REST, um serviço pode ter várias operações, e estas também podem ser chamadas de recursos;
- Na implementação da especificação JAX-RS (JSR 339), feita aqui pelo Jersey, o serviço é implementado por uma classe Java;
- A operação do serviço é implementada por um método dessa classe;
- Quando uma requisição HTTP tenta acessar um

recurso, como uma operação de um serviço REST, ela é roteada até chegar no método que vai tratá-la, método esse que pertence a uma classe que representa o serviço em questão.

Veja o que deve ser feito para utilizar anotações nos métodos e implementar os requisitos apresentados de segurança no serviço de produtos:

1. Na classe `com.siecola.exemplo1.authentication.AuthFilter` , adicione o atributo privado a seguir para acessar informações do recurso (serviço) e suas operações:

```
@Context
private ResourceInfo resourceInfo;
```

   Por enquanto, o que será feito é somente saber se o método destino da requisição possui alguma anotação.

2. No início do método `filter` , coloque a linha a seguir para poder ter acesso às informações do método destino da requisição:

```
Method method = resourceInfo.getResourceMethod();
```

   A chamada `resourceInfo.getResourceMethod()` retorna um objeto do tipo `Method` , que traz informações sobre o método que implementa a operação de destino da requisição.

3. Logo em seguida, usando a variável `method` , verifique se o método a ser acessado pela requisição possui a anotação `@PermitAll` , que faz com que a requisição seja autorizada, sem a necessidade da autenticação do usuário:

```
if (method.isAnnotationPresent(PermitAll.class)) {
    return;
}
```

Com a chamada `method.isAnnotationPresent()` , pode-se verificar a presença de uma anotação no método invocado pela requisição. Dessa forma, é possível tomar decisões no mecanismo de autenticação e autorização, como é o caso aqui, que verifica a presença da anotação `@PermitAll` . Se ela existir, não há a necessidade de continuar executando o método `filter` para verificar as credenciais de acesso, fazendo com que a requisição siga adiante até o seu destino.

Isso pode ser utilizado para permitir o acesso a operações de serviços sem a necessidade de autenticação do usuário.

4. Agora, para fazer com que as operações de listar todos os produtos e de buscar somente um pelo seu código fiquem abertas, sem a necessidade de autenticação, basta colocar a anotação `@PermitAll` nos seus respectivos métodos, como da forma a seguir:

    - Para o método de ler somente um produto pelo seu código, acessado pela URL `api/products/{code}` , as anotações devem ficar assim:

    ```
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @Path("/{code}")
    @PermitAll
    public Product getProduct(@PathParam("code") int code)
    {
    ```

    - E para o método de ler todos os produtos, acessado pela URL `api/products/` , devem ficar da seguinte forma:

    ```
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @PermitAll
    public List<Product> getProducts() {
    ```

    Para certificar-se de que tudo está funcionando, execute os seguintes testes, sem configurar o cabeçalho

`Authorization` com as credenciais de acesso no REST Console, e verifique os resultados:

- Acesse a operação de listar todos os produtos por meio da URL `api/products` : você verá que a lista de produtos será respondida pela operação;

- Busque um produto existente pelo seu código através da URL `api/products/{code}` : você terá o produto informado na resposta.

- Acesse a operação para excluir o produto pelo seu código através da mesma URL `api/products/{code}` : a aplicação negará sua requisição, com o código `HTTP 401 UNAUTHORIZED` , pois essa é uma operação que necessita que um usuário seja autenticado para que ela possa ser executada.

  Veja que a primeira diretriz de segurança do serviço de produtos, definida no início desta seção, já foi satisfeita.

5. Para implementar as demais regras de autorização do serviço de produtos, é necessário fazer pequenas mudanças estruturais no método `filter` da classe `com.siecola.exemplo1.authentication.AuthFilter` , apenas para ficar um código mais elegante e de fácil entendimento. A primeira é tirar a parte que valida as credenciais de acesso, passando para um método privado, que também terá a função de checar se papel do usuário tem permissão especificada no método, através de suas anotações, como será visto mais adiante. Veja como deve ficar esse novo método privado:

```
private boolean checkCredentialsAndRoles (String username,
String password, Set<String> roles) {
    boolean isUserAllowed = false;
```

```
        if (username.equals("Admin") && password.equals("Admin"
)) {
            if(roles.contains("ADMIN"))
            {
                isUserAllowed = true;
            }
        }

        if (isUserAllowed == false) {
            if (username.equals("User") && password.equals("Use
r")) {
                if(roles.contains("USER"))
                {
                    isUserAllowed = true;
                }
            }
        }

        return isUserAllowed;
 }
```

Como a ideia é fazer com que exista dois usuários com papéis distintos, `ADMIN` e `USER`, e ainda não há base de dados de usuários, é necessário realizar os testes das credenciais e dos papéis separadamente. Quando o serviço de usuários e sua base de dados forem construídos no capítulo *Adicionando o serviço de usuários*, esse método ficará mais simples e elegante.

Repare que agora, além de verificar o usuário e a senha, o método também verifica se o usuário pertence a um dos papéis permitidos para a requisição. Essa lista de papéis pode ser colocada na anotação `@RolesAllowed`, no método que implementa a operação do serviço.

Porém, da forma como foi implementado o método `checkCredentialsAndRoles`, todas as operações deverão possuir uma lista de papéis de usuários que podem acessá-las. Isso também pode ser alterado. Por exemplo, se a lista informada no parâmetro estiver vazia, nenhum teste é feito para o papel do usuário, liberando ou não somente com a

verificação das credenciais.

6. Agora, no final do método `filter`, no lugar onde as credenciais do usuário eram verificadas, retire o trecho de código a seguir, que fazia a validação das credenciais que estavam fixas no código:

```
if (loginPassword[0].equals("Admin")
     && loginPassword[1].equals("Admin")) {
 return;
} else {
 requestContext.abortWith(Response.status(Response.Status.UN
AUTHORIZED)
         .entity(ACCESS_UNAUTHORIZED).build());
}
```

E substitua pela chamada do novo método privado, criado no passo anterior (`checkCredentialsAndRoles`), buscando antes a lista de papéis que está anotada no método:

```
RolesAllowed rolesAllowed = method.getAnnotation(RolesAllowe
d.class);
Set<String> rolesSet = new HashSet<String>(Arrays.asList(rol
esAllowed.value()));

if (checkCredentialsAndRoles (loginPassword[0], loginPasswor
d[1], rolesSet) == false) {
 requestContext.abortWith(Response.status(Response.Status.UN
AUTHORIZED)
         .entity(ACCESS_UNAUTHORIZED).build());
 return;
}
```

As duas primeiras linhas desse trecho de código resgatam a lista de papéis de usuário que o método destino da requisição permite, através da anotação `@RolesAllowed()` , que ainda será colocada nele.

Logo a seguir, o método `checkCredentialsAndRoles` é chamado, passando as credenciais fornecidas na requisição e também a lista de papéis que o método permite. Se as credenciais estiverem corretas e o usuário possuir um papel

presente nessa lista, a requisição será aceita e encaminhada para ser trata pelo método responsável por isso; caso contrário, será abortada com o código `HTTP 401 UNAUTHORIZED` , como já era feito.

7. Por último, deve-se colocar as anotações nos métodos das operações restantes do serviço de produtos na classe `com.siecola.exemplo1.services.ProductManager` . Lembrando de que as diretrizes de segurança que ainda faltam são:

    - Permitir que somente o usuário com papel `ADMIN` possa apagar um produto;

    - O cadastro e alteração de produtos deve ser feito somente por usuários com o papel `USER` .

    Para satisfazer a primeira regra, basta colocar a anotação `@RolesAllowed("ADMIN")` no método `deleteProduct` , como mostrado a seguir:

```
@DELETE
@Produces(MediaType.APPLICATION_JSON)
@Path("/{code}")
@RolesAllowed("ADMIN")
public Product deleteProduct(@PathParam("code") int cod
e) {
```

    Isso fará com que, no método `filter` da classe `com.siecola.exemplo1.authentication.AuthFilter` , a lista de papéis recuperada para validação do usuário tenha apenas o valor `ADMIN` . Ou seja, somente um usuário com esse papel poderá acessar essa operação desse serviço.

    Para satisfazer a segunda regra, da alteração e criação de produtos somente pelo usuário com papel `USER` , basta colocar uma anotação da mesma forma, porém com o

valor `USER` .

Para a operação de criação de produtos:

```
@POST
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
@RolesAllowed("USER")
public Product saveProduct(Product product) {
```

E para a operação de alteração de produtos:

```
@PUT
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
@Path("/{code}")
@RolesAllowed("USER")
public Product alterProduct(@PathParam("code") int code
, Product product) {
```

Pronto! O serviço de gerenciamento de produtos agora possui autenticação e autorização de usuários, inclusive com definição de diferentes papéis.

Realize alguns testes, acessando as operações desse serviço usando o REST Console, alternando entre o usuário `Admin` e o `User` para ver os diferentes comportamentos introduzidos pelas alterações realizadas nesta seção, principalmente com relação aos papéis.

Múltiplos papéis para uma mesma operação

Durante seus testes com as novas regras de autorização de usuários no serviço de produtos, você deve ter percebido algo estranho: o usuário com papel `ADMIN` não pode cadastrar e alterar produtos. A explicação, na verdade, é bem simples, pois seu papel não está presente na anotação nesses métodos.

Para resolver isso, basta alterar a anotação `@RolesAllowed` para ficar da seguinte forma:

```
@RolesAllowed({"ADMIN", "USER"})
```

Isso fará com que usuários os papéis `ADMIN` e `USER` possam ter acessos às operações de cadastramento e alteração de produtos.

Testando no GAE

Publique a aplicação no GAE e faça os testes com os dois usuários `Admin` e `User` . Veja as mensagens de log no console do GAE quando você tenta acessar alguma operação que o usuário não tem permissão. Perceba que a chamada do método:

```
requestContext.abortWith(Response.status(Response.Status.UNAUTHORI
ZED)
        .entity(ACCESS_UNAUTHORIZED).build());
```

faz com que uma mensagem de log com o código `HTTP 401 UNAUTHORIZED` apareça no console, indicando que aquela requisição não foi aceita pelo aplicação. Veja também que a mensagem de texto que aparece ao usuário também fica registrada no log, para que você possa ter mais detalhes do que aconteceu.

Você pode encontrar o código completo do projeto `exemplo1` no repositório: https://github.com/siecola/GAEBook.

8.6 CONCLUSÃO

Neste capítulo, você aprendeu como proteger serviços REST com o mecanismo `HTTP Basic Authentication` por meio da construção de um filtro, onde tudo acontece:

- Leitura do cabeçalho `Authorization` ;
- Leitura das credenciais de acesso enviadas na requisição;
- Leitura das anotações dos métodos que implementam as operações;
- Validação das credenciais e do papel do usuário.

A partir de agora, qualquer serviço que for criado na aplicação `exemplo1` poderá utilizar o mesmo mecanismo já desenvolvido, utilizando as anotações que foram colocadas no serviço de produtos.

No próximo capítulo, você tornará o mecanismo de autenticação ainda mais eficiente e sofisticado, criando um serviço de gerenciamento de usuários do sistema e uma tabela no Datastore para poder armazenar seus dados. Com essa tabela, será possível guardar as credenciais de acesso, papéis e outras informações sobre o usuário.

CAPÍTULO 9

ADICIONANDO O SERVIÇO DE USUÁRIOS

Neste capítulo, você incrementará a aplicação `exemplo1` , adicionando mais um serviço que será muito importante para tornar o mecanismo de autenticação mais profissional. Junto com esse novo serviço, será criado um novo tipo (*Kind*) no Google Cloud Datastore, que será o usuário, muito útil para a funcionalidade de enviar mensagens a dispositivos móveis com o Google Cloud Messaging, que será visto no próximo capítulo, *Enviando mensagens com o Google Cloud Messaging*.

9.1 CRIANDO O MODELO DE USUÁRIOS

Antes de criar o novo serviço, é necessário criar o modelo de dados que o representa. Ele também será utilizado para a construção novo tipo no Google Cloud Datastore.

O modelo de usuários deve ter, pelo menos, os seguintes atributos:

- Identificação única no Google Cloud Datastore;
- Registro no Google Cloud Messaging;
- E-mail do usuário, para ser utilizado como login;
- Senha de acesso do usuário;
- Data e hora do último login;
- Data e hora do último registro no Google Cloud

Messaging.

O atributo com o valor de registro no Google Cloud Messaging será utilizado para enviar mensagens aos dispositivos móveis, registrados para receber notificações da aplicação. Os detalhes da implementação esse mecanismo será mostrado no capítulo *Enviando mensagens com o Google Cloud Messaging*.

De posse dos atributos do usuário, você pode criar a classe modelo `User` para representá-lo no pacote `com.siecola.exemplo1.models` . Veja no código a seguir como ela pode ficar:

```
public class User {
    private long id;
    private String email;
    private String password;
    private String gcmRegId;
    private Date lastLogin;
    private Date lastGCMRegister;
    private String role;
}
```

Essa classe modelo também será usada para persistir os dados dos usuários em um novo tipo, ou tabela se você preferir assim chamar no Google Cloud Datastore.

9.2 CRIANDO O SERVIÇO DE USUÁRIOS

Para criar o serviço de usuário, é necessário antes planejar alguns requisitos, como qual será a URL de acesso, quais operações ele deverá ter e quais as permissões de acesso a essas operações:

- **Requisito 1**: a URL base do serviço deverá ser `api/users` .
- **Requisito 2**: deverá ter uma operação para listar todos os usuários, que poderá ser acessada somente por usuários com o papel `ADMIN` .

- **Requisito 3**: a operação para cadastrar novos usuários poderá ser feita por qualquer usuário. Somente o usuário com papel `ADMIN` poderá cadastrar outro com papel `ADMIN`. Não poderão existir usuários com e-mails repetidos.
- **Requisito 4**: a operação para alterar um usuário poderá ser feita somente por um usuário `ADMIN` ou pelo próprio usuário. Nesse último caso, ele não poderá alterar o papel (role). O e-mail do usuário deverá ser usado como parâmetro da operação para localizá-lo.
- **Requisito 5**: a operação para retornar todas as informações de um usuário só poderá ser feita por um usuário `ADMIN` ou pelo próprio usuário. O e-mail do usuário deverá ser utilizado como parâmetro da operação para localizá-lo.
- **Requisito 6**: a operação para excluir um usuário só poderá ser feita por um usuário `ADMIN` ou pelo próprio usuário. O e-mail do usuário deverá ser usado como parâmetro da operação para localizá-lo.
- **Requisito 7**: deverá possuir uma operação para atualização do valor do registro no Google Cloud Messaging, que receberá o valor desse registro. Tal operação só poderá ser acessada pelo próprio usuário. Sua URL de acesso deverá ser `api/users/update_gcm_reg_id/{gcmRegId}`.
- **Requisito 8**: o e-mail e senha de acesso do usuário não poderão ser vazios e isso deve ser verificado nas operações de cadastro e alteração;

Algumas coisas novas e interessantes vão aparecer, em toda a aplicação, com a utilização do serviço e das entidades de usuário, por exemplo:

- Possibilidade de conhecer o usuário que está acessando

uma operação, de dentro do método da classe que a implementa;

- Validação de campos do modelo nas operações de cadastro e alteração;
- Limitação de acesso a recursos somente pelo usuário que é dono ou tem direito a ele.

Criando a classe do serviço de usuários

Para começar, deve-se criar a classe que representará o serviço de usuários, com o nome de `UserManager` no pacote `com.siecola.exemplo1.services`. Para já satisfazer o **Requisito 1**, que é o endereço base da operação, coloque a anotação `@Path` com a URL sugerida:

```
@Path("/users")
public class UserManager {

    private static final Logger log = Logger.getLogger("UserManage
r");

    public static final String USER_KIND = "Users";

    public static final String PROP_EMAIL = "email";
    public static final String PROP_PASSWORD = "password";
    public static final String PROP_GCM_REG_ID = "gcmRegId";
    public static final String PROP_LAST_LOGIN = "lastLogin";
    public static final String PROP_LAST_GCM_REGISTER = "lastGCMRe
gister";
    public static final String PROP_ROLE = "role";

}
```

O atributo estático `Logger log` vai servir para gerar mensagens de log quando necessário e se você quiser. Já o `USER_KIND` representa o nome do tipo a ser criado no Datastore. Os demais atributos estáticos são os nomes das propriedades do tipo `Users`.

Como foi feito no serviço de produtos, é interessante criar os

métodos privados para conversão da entidade do Datastore para o objeto `User` , a ser utilizado como parâmetro de entrada e saída das operações (e vice-e-versa). Dessa forma, você pode isolar a lógica das operações com mais clareza, deixando apenas as questões de interação com o Datastore.

Para isso, crie o método que faz a conversão do objeto `User` para o tipo da entidade a ser armazenada no Datastore:

```
private void userToEntity (User user, Entity userEntity) {
    userEntity.setProperty(PROP_EMAIL, user.getEmail());
    userEntity.setProperty(PROP_PASSWORD, user.getPassword());
    userEntity.setProperty(PROP_GCM_REG_ID, user.getGcmRegId());
    userEntity.setProperty(PROP_LAST_LOGIN, user.getLastLogin());
    userEntity.setProperty(PROP_LAST_GCM_REGISTER, user.getLastGCM
Register());
    userEntity.setProperty(PROP_ROLE, user.getRole());
}
```

Em seguida, crie o método privado para fazer a conversão contrária, ou seja, de uma entidade `Users` do Datastore para um objeto `User` :

```
private User entityToUser (Entity userEntity) {
    User user = new User();
    user.setId(userEntity.getKey().getId());

    user.setEmail((String) userEntity.getProperty(PROP_EMAIL));
    user.setPassword((String) userEntity.getProperty(PROP_PASSWORD
));
    user.setGcmRegId((String) userEntity.getProperty(PROP_GCM_REG_
ID));
    user.setId(userEntity.getKey().getId());
    user.setLastLogin((Date) userEntity.getProperty(PROP_LAST_LOGI
N));
    user.setLastGCMRegister((Date) userEntity.getProperty(PROP_LAS
T_GCM_REGISTER));
    user.setRole((String) userEntity.getProperty(PROP_ROLE));

    return user;
}
```

Inicialização da aplicação

Se você reparar bem nos requisitos descritos no início desta seção, você vai perceber que há um problema técnico: como cadastrar um usuário, se essa operação só pode ser feita por um usuário com o papel `ADMIN`? Então como cadastrarei o primeiro usuário?

Para resolver isso, é necessário criar um servlet de inicialização, que será chamado assim que a aplicação for iniciada. Assim, será possível criar um primeiro usuário na aplicação caso não exista nenhum criado ainda. Obviamente, esse usuário deverá ter o papel `ADMIN` com uma senha conhecida. Quando a aplicação for instalada e executada pela primeira vez, essa senha do usuário `ADMIN` poderá ser alterada.

Para ficar melhor ainda e evitar que todos os usuários `ADMIN` sejam apagados, pode-se fazer uma lógica mais elaborada nesse servlet de inicialização, contando quantos usuários com o papel `ADMIN` existem e, caso não haja nenhum, um novo é criado com as credenciais conhecidas. Essa técnica é muito utilizada quando há essa necessidade de algo somente funcionar, se já houver dados válidos em uma tabela do banco de dados, por exemplo.

Para criar esse novo servlet e fazer com que ele seja chamado na inicialização da aplicação, deve ser adicionada uma nova configuração no arquivo `src/main/webapp/WEB-INF/web.xml`, pela tag `<listener>`. Dentro dessa tag, deverá ser informada a classe com seu caminho completo:

```
<listener>
    <listener-class>com.siecola.exemplo1.InitServletContextClass</
listener-class>
</listener>
```

Agora, no pacote `com.siecola.exemplo1`, crie a classe `InitServletContextClass` implementando a interface `ServletContextListener`:

```java
public class InitServletContextClass implements ServletContextList
ener {

    private static final Logger log = Logger.getLogger("InitServle
tContextClass");

    @Override
    public void contextInitialized(ServletContextEvent arg0) {
        log.info("Aplicação Exemplo1 iniciada!");
    }

    @Override
    public void contextDestroyed(ServletContextEvent arg0) {
    }
}
```

> O método `contextInitialized` será chamado na inicialização da aplicação. É importante de lembrar de que, no ambiente de desenvolvimento, esse método será invocado assim que a aplicação entrar em execução no Eclipse. Porém, no ambiente de produção do GAE, ele só será chamado quando algum serviço da aplicação for invocado.

Para deixar o código mais simples, crie o método privado `initializeUserEntities` para ser chamado de dentro de `contextInitialized` :

```java
@Override
public void contextInitialized(ServletContextEvent arg0) {
    log.info("Aplicação Exemplo1 iniciada!");

    initializeUserEntities();
}
```

Ele será o responsável por realizar a lógica de buscar no Google Cloud Datastore por entidades do tipo `Users` com o papel `ADMIN` ; e caso não encontre nenhum, deverá criar com as credenciais de acesso conhecidas, como mostra o trecho de código a seguir:

```
private void initializeUserEntities() {
    DatastoreService datastore = DatastoreServiceFactory
            .getDatastoreService();

    Filter roleFilter = new FilterPredicate(UserManager.PROP_ROLE,
 FilterOperator.EQUAL, "ADMIN");

    Query query = new Query(UserManager.USER_KIND).setFilter(roleF
ilter);
    List <Entity> entities = datastore.prepare(query).asList(Fetch
Options.Builder.withLimit(1));

    if (entities.size() == 0) {
        log.info("Nenhum usuário encontrado. Inicializando o tipo
Users no Datastore");

        Key userKey = KeyFactory.createKey(UserManager.USER_KIND,
"userKey");
        Entity userEntity = new Entity(UserManager.USER_KIND, user
Key);

        userEntity.setProperty(UserManager.PROP_EMAIL, "admin@siec
ola.com");
        userEntity.setProperty(UserManager.PROP_PASSWORD, "Admin#7"
);
        userEntity.setProperty(UserManager.PROP_GCM_REG_ID, "");
        userEntity.setProperty(UserManager.PROP_LAST_LOGIN, Calend
ar.getInstance().getTime());
        userEntity.setProperty(UserManager.PROP_LAST_GCM_REGISTER,
 Calendar.getInstance().getTime());
        userEntity.setProperty(UserManager.PROP_ROLE, "ADMIN");


        datastore.put(userEntity);
    }
}
```

Repare que, no início do método, um filtro é criado para pesquisar por entidades do tipo `Users` que possuam a propriedade de nome `role` igual a `ADMIN`. Em seguida, é feita uma consulta para saber se existe pelo menos uma entidade que satisfaça tal condição. Caso não exista nenhuma entidade `Users` com a propriedade `role` igual a `ADMIN`, é criado então uma nova entidade, com as credenciais de acesso fixas. Se esse código executar uma vez, na próxima vez que a aplicação entrar em execução, esse

usuário não será criado, a não ser que ele tenha sido apagado ou tenha seu papel alterado.

Há algumas outras formas de se pesquisar simplesmente pela existência de entidades no Datastore, como se fosse contá-las, mas a sintaxe é um pouco complexa comparada à redução de processamento que ela traz. Para esse caso, que só será executado uma vez, quando a aplicação iniciar, não vale a pena.

Execute a sua aplicação na máquina de desenvolvimento e verifique se, no console de administração, no endereço `http://localhost:8888/_ah/admin`, o tipo `Users` foi criado e a entidade criada pelo método `initializeUserEntities` realmente aparece

Alterando o mecanismo de autenticação dos serviços

Agora que há pelo menos um usuário no Datastore, você pode alterar o mecanismo de autenticação e autorização dos serviços da aplicação `exemplo1` para utilizar os usuários que estão cadastrados lá.

Lembre-se de que no capítulo *Protegendo serviços com HTTP Basic Authentication*, a classe `com.siecola.exemplo1.authentication.AuthFilter` utilizou credenciais de acesso fixas no código. Agora é o momento de alterar seus métodos e deixá-los mais elegantes e profissionais.

Para isso, abra essa classe, vá ao método `checkCredentialsAndRoles` e faça as seguintes alterações:

1. Utilize a informação do `username`, passada como parâmetro para o método, para localizar o usuário no Datastore;
2. Caso não encontre, retorne dizendo que o usuário não está autorizado;

3. Se o usuário existir, confirme se a senha está correta e que ele é de um dos papéis autorizados para acessar a operação do serviço em questão. Caso a senha não bata ou ele não seja de um dos papéis autorizados, retorne negando a autorização;
4. Passe mais um parâmetro para o método, que é a variável `ContainerRequestContext requestContext` , parâmetro do método `filter` , para que ela seja preenchida com as informações do usuário, caso ele seja autorizado. Essa informação será passada para dentro do método que trata a operação do serviço para obter informações como `username` ou o e-mail, como está sendo usado.

Para começar, altere a assinatura do método para receber como parâmetro a variável `ContainerRequestContext requestContext` :

```
private boolean checkCredentialsAndRoles (String username, String
password, Set<String> roles, ContainerRequestContext requestContex
t)
```

A chamada desse método deve ficar assim:

```
if (checkCredentialsAndRoles (loginPassword[0], loginPassword[1],
rolesSet, requestContext) == false) {
    requestContext.abortWith(Response.status(Response.Status.UNAUT
HORIZED)
            .entity(ACCESS_UNAUTHORIZED).build());
    return;
}
```

Dentro do método `checkCredentialsAndRoles` , o objeto do tipo `User` será inserido na variável `requestContext` , mas para isso é necessário fazer com a classe `User` implemente a interface `Principal` :

```
public class User implements Principal {

    private long id;
    private String email;
    private String password;
```

```java
    private String gcmRegId;
    private Date lastLogin;
    private Date lastGCMRegister;
    private String role;

    @Override
    public String getName() {
        return this.email;
    }

    //getters and setters
}
```

O método `checkCredentialsAndRoles` deverá ficar como mostra o código a seguir. Apesar de ele parecer um pouco mais complexo, agora ele faz seu trabalho de checar se um usuário está ou não autorizado a acessar uma operação, utilizando a informação obtida do Datastore, por meio de seu e-mail.

```java
private boolean checkCredentialsAndRoles (String username, String
password, Set<String> roles, ContainerRequestContext requestContex
t) {
    boolean isUserAllowed = false;

    DatastoreService datastore = DatastoreServiceFactory
            .getDatastoreService();

    Filter emailFilter = new FilterPredicate(UserManager.PROP_EMAI
L, FilterOperator.EQUAL, username);

    Query query = new Query(UserManager.USER_KIND).setFilter(email
Filter);
    Entity userEntity = datastore.prepare(query).asSingleEntity();

    if (userEntity != null) {
        if (password.equals(userEntity.getProperty(UserManager.PRO
P_PASSWORD)) &&
                roles.contains(userEntity.getProperty(UserManager.
PROP_ROLE))) {

            final User user = new User();

            user.setEmail((String) userEntity.getProperty(UserMana
ger.PROP_EMAIL));
            user.setPassword((String) userEntity.getProperty(UserM
anager.PROP_PASSWORD));
```

```
            user.setId(userEntity.getKey().getId());
            user.setGcmRegId((String) userEntity.getProperty(UserM
anager.PROP_GCM_REG_ID));
            user.setLastLogin((Date) Calendar.getInstance().getTim
e());
            user.setLastGCMRegister((Date) userEntity.getProperty(
UserManager.PROP_LAST_GCM_REGISTER));
            user.setRole((String) userEntity.getProperty(UserManag
er.PROP_ROLE));

            userEntity.setProperty(UserManager.PROP_LAST_LOGIN, us
er.getLastLogin());
            datastore.put(userEntity);

            requestContext.setSecurityContext(new SecurityContext(
) {

                @Override
                public boolean isUserInRole(String role) {
                    return role.equals(user.getRole());
                }

                @Override
                public boolean isSecure() {
                    return true;
                }

                @Override
                public Principal getUserPrincipal() {
                    return user;
                }

                @Override
                public String getAuthenticationScheme() {
                    return SecurityContext.BASIC_AUTH;
                }
            });

            isUserAllowed = true;
        }
    }

    return isUserAllowed;
}
```

Repare que, depois que as credenciais de acesso do usuário são verificadas e se ele possui autorização através de seu papel, o objeto

do tipo `User` é inserido no contexto da requisição pela chamada `requestContext.setSecurityContext`. Essa informação poderá ser usada dentro do método que implementa a operação do serviço, por exemplo, para verificar qual é o usuário que realizou a requisição. Isso é algo muito importante para verificar se um usuário está realmente acessando uma informação que pertence a ele e não a outro usuário, por exemplo.

Agora o mecanismo de autenticação está mais elaborado e elegante. Você verá como ficará fácil implementar as regras de autorização por meio das anotações nos métodos dos serviços e também verificando qual usuário está acessando.

Criando a operação para listar todos os usuários

Seguindo com a implementação do **Requisito 2**, que é a operação para listar todos os usuários, volte na classe `com.siecola.exemplo1.services.UserManager` e crie o método público para buscar no Datastore todas as entidades do tipo `Users`. Lembre-se de que o requisito pede que essa operação só possa ser acessada por um usuário com papel `ADMIN`, por isso ela deve ser anotada para tal:

```
@GET
@Produces(MediaType.APPLICATION_JSON)
@RolesAllowed({"ADMIN"})
public List<User> getUsers() {

    List<User> users = new ArrayList<>();
    DatastoreService datastore = DatastoreServiceFactory
            .getDatastoreService();

    Query query = new Query(USER_KIND).addSort(PROP_EMAIL, SortDir
ection.ASCENDING);

    List<Entity> userEntities = datastore.prepare(query).asList(
            FetchOptions.Builder.withDefaults());

    for (Entity userEntity : userEntities) {
        User user = entityToUser(userEntity);
```

```
        users.add(user);
    }

    return users;
}
```

Repare que o objeto do tipo `Query` é instanciado para buscar as entidades do tipo `Users` , ordenada de forma crescente (ou alfabética) pelo valor da propriedade e-mail.

A anotação `@RolesAllowed({"ADMIN"})` é para fazer com que essa operação seja acessada somente por usuários com o papel `ADMIN` , como já foi dito.

Criando a operação de recuperar um usuário

A implementação do **Requisito 5** é semelhante ao método anterior, mas em vez de retornar uma lista, o método deve receber o e-mail como parâmetro e pesquisá-lo no Datastore.

Essa operação deve ser acessada somente por um usuário com o papel `ADMIN` ou pelo próprio usuário dono das informações. Porém, colocar a anotação `@RolesAllowed({"ADMIN", "USER"})` não resolve, já que faria com que qualquer usuário com papel `USER` acesse, e não somente o dono. Será necessário implementar uma lógica para verificar se a requisição foi feita pelo usuário que é dono das informações, e isso será feito pegando a informação que foi injetada na variável `ContainerRequestContext requestContext` do método `filter` da classe `AuthFilter` , ou seja, o usuário em si.

Veja como fica o método:

```
@GET
@Produces(MediaType.APPLICATION_JSON)
@RolesAllowed({"ADMIN", "USER"})
@Path("/{email}")
public User getUser(@PathParam(PROP_EMAIL) String email) {
```

```
    if (securityContext.getUserPrincipal().getName().equals(email)
 ||
            securityContext.isUserInRole("ADMIN")) {
        DatastoreService datastore = DatastoreServiceFactory
                .getDatastoreService();

        Filter emailFilter = new FilterPredicate(PROP_EMAIL, Filte
rOperator.EQUAL, email);

        Query query = new Query(USER_KIND).setFilter(emailFilter);
        Entity userEntity = datastore.prepare(query).asSingleEntit
y();

        if (userEntity != null) {
            User user = entityToUser(userEntity);

            return user;
        } else {
            throw new WebApplicationException(Status.NOT_FOUND);
        }
    } else {
        throw new WebApplicationException(Status.FORBIDDEN);
    }
}
```

Você pode se perguntar se a seguinte instrução não daria exceção se houvesse mais de uma entidade com o mesmo e-mail:

```
Entity userEntity = datastore.prepare(query).asSingleEntity();
```

Sim, daria, mas isso será resolvido nas validações nas operações de inserir e alterar usuários.

Repare no teste feito logo no início do método:

```
if (securityContext.getUserPrincipal().getName().equals(email) ||
        securityContext.isUserInRole("ADMIN")) {
```

Ele serve justamente para permitir que o recurso seja acessado somente por usuários com o papel `ADMIN` ou pelo próprio dono da informação. Até poderia se dizer que a anotação `@RolesAllowed` fica desnecessária, mas pode existir um outro tipo de papel, que talvez não tenha permissão para acessar essa operação, então é

melhor pecar pelo excesso aqui.

O atributo `securityContext` deve ser declarado da seguinte forma:

```
@Context
SecurityContext securityContext;
```

Ele traz as informações da requisição, inclusive do usuário que foi inserido no método `checkCredentialsAndRoles` da classe `AuthFilter` .

Escrevendo a operação para criar um usuário

De acordo com o **Requisito 3**, a operação de criação de usuários pode ser feita por qualquer usuário, mas somente aquele que tiver o papel `ADMIN` poderá criar outro com esse mesmo papel. Isso pode ser feito, verificando se há um usuário autenticado na requisição, e se ele possui o papel `ADMIN` por meio do teste `securityContext.isUserInRole("ADMIN")` , que retorna verdadeiro caso o usuário esteja autenticado pertença a esse papel.

Além disso, é necessário verificar se já não existe um usuário com o mesmo endereço de e-mail, para não haver usuários duplicados no Datastore. Para isso, pode-se criar um método privado para fazer essa verificação, como mostra o código a seguir:

```
private boolean checkIfEmailExist (User user) {
    DatastoreService datastore = DatastoreServiceFactory
            .getDatastoreService();

    Filter emailFilter = new FilterPredicate(PROP_EMAIL, FilterOpe
rator.EQUAL, user.getEmail());

    Query query = new Query(USER_KIND).setFilter(emailFilter);
    Entity userEntity = datastore.prepare(query).asSingleEntity();

    if (userEntity == null) {
        return false;
    } else {
```

```
        if (userEntity.getKey().getId() == user.getId()) {
            //está alterando o mesmo user
            return false;
        } else {
            return true;
        }
    }
}
```

Esse método é muito semelhante ao código do método `checkIfCodeExist` da classe `ProductManager`, porém a validação aqui é feita com o e-mail do usuário.

O método para criar um usuário deve ficar da seguinte forma:

```
@POST
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
@PermitAll
public User saveUser(User user) {

    DatastoreService datastore = DatastoreServiceFactory
            .getDatastoreService();

    if (!checkIfEmailExist (user)) {
        if (!securityContext.isUserInRole("ADMIN")) {
            user.setRole("USER");
        }
        Key userKey = KeyFactory.createKey(USER_KIND, "userKey");
        Entity userEntity = new Entity(USER_KIND, userKey);

        user.setGcmRegId("");
        user.setLastGCMRegister(null);
        user.setLastLogin(null);

        userToEntity (user, userEntity);

        datastore.put(userEntity);

        user.setId(userEntity.getKey().getId());
    } else {
        throw new WebApplicationException("Já existe um usuário ca
dastrado com o mesmo e-mail", Status.BAD_REQUEST);
    }

    return user;
}
```

Veja que, como a operação pode ser acessada por qualquer usuário, a anotação `@PermitAll` deve ser colocada. A validação de segurança fica por conta do teste se o usuário autenticado possui o papel `ADMIN` para permiti-lo criar outro usuário com o mesmo papel.

Validando o modelo de usuário

Tanto na operação de criação quanto alteração de usuários, é necessário fazer algumas validações, como:

- Se o valor do campo e-mail está dentro do padrão de endereços de e-mails;
- Se a senha não está vazia;
- Se o papel não está vazio.

Você poderia escrever um método privado para fazer essas validações ou utilizar algumas anotações na classe modelo do usuário, como mostra o trecho a seguir, fazendo exatamente o que foi listado:

```
public class User implements Principal {

    private long id;

    @Email
    private String email;

    @NotNull
    private String password;

    private String gcmRegId;

    private Date lastLogin;

    private Date lastGCMRegister;

    @NotNull
    private String role;

    @Override
```

```
    public String getName() {
        return this.email;
    }

    //getters and setters
}
```

A anotação `@Email` garante que o valor do campo por ele anotado tenha o padrão de um endereço de e-mail. Já a anotação `@NotNull` faz com que o campo tenha de ter um valor diferente de `null`.

Para utilizar essas anotações, basta adicionar mais uma dependência no arquivo `pom.xml`:

```
<dependency>
    <groupId>org.glassfish.jersey.ext</groupId>
    <artifactId>jersey-bean-validation</artifactId>
    <version>2.22.1</version>
</dependency>
```

E para fazer com que a validação seja de fato aplicada nos métodos das operações dos serviços, basta utilizar a anotação `@Valid` antes do parâmetro nos métodos, como o exemplo a seguir para o método de criação de um novo usuário:

```
public User saveUser(@Valid User user) {
```

Agora, todo usuário será validado antes de ser passado para o método `saveUser`, e caso esteja inválido, de acordo com as anotações na classe `User`, a requisição será negada com o código `HTTP 400 Bad Request`.

Isso garante o que pede o **Requisito 8**.

Criando a operação de alterar um usuário

No **Requisito 4**, a operação de alterar um usuário é semelhante ao da criação, com algumas validações e regras a mais, como:

- Somente o próprio usuário ou outro com papel `ADMIN` podem alterar os dados;
- O papel não pode ser alterado se o usuário for `USER` ;

Isso tudo pode ser feito com testes que já foram usados aqui ou no serviço de produtos. Veja como ele pode ficar:

```
@PUT
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
@Path("/{email}")
@RolesAllowed({"ADMIN", "USER"})
public User alterUser(@PathParam("email") String email, @Valid Use
r user) {
    if (user.getId() != 0) {
        if (securityContext.getUserPrincipal().getName().equals(em
ail) ||
                securityContext.isUserInRole("ADMIN")) {
            if (!checkIfEmailExist (user)) {
                DatastoreService datastore = DatastoreServiceFacto
ry
                        .getDatastoreService();

                Filter emailFilter = new FilterPredicate(PROP_EMAI
L,
                        FilterOperator.EQUAL, email);

                Query query = new Query(USER_KIND).setFilter(email
Filter);

                Entity userEntity = datastore.prepare(query).asSin
gleEntity();

                if (userEntity != null) {
                    userToEntity (user, userEntity);

                    if (!securityContext.isUserInRole("ADMIN")) {
                        user.setRole("USER");
                    }
                    datastore.put(userEntity);

                    return user;
                } else {
                    throw new WebApplicationException(Status.NOT_F
OUND);
                }
```

```
            } else {
                throw new WebApplicationException("Já existe um us
uário cadastrado com o mesmo e-mail", Status.BAD_REQUEST);
            }
        } else {
            throw new WebApplicationException(Status.FORBIDDEN);
        }
    } else {
        throw new WebApplicationException("O ID do usuário deve se
r informado para ser alterado", Status.BAD_REQUEST);
    }
}
```

Essa operação ainda poderia ter outras validações, como por exemplo, evitar que o usuário, se não for `ADMIN`, não poder alterar os campos `gcmRegId`, `lastLogin` e `lastGCMRegister`.

Criando a operação para excluir um usuário

Agora que você já sabe como aplicar regras de validações e autorizações, fica fácil implementar a operação para apagar um usuário, como pede o **Requisito 6**. Ele pede que um usuário só possa ser apagado por ele mesmo ou por um outro que seja `ADMIN`. Além disso, a operação deve receber como parâmetro o e-mail do usuário. Veja como ela pode ficar:

```
@DELETE
@Produces(MediaType.APPLICATION_JSON)
@Path("/{email}")
@RolesAllowed({"ADMIN", "USER"})
public User deleteUser(@PathParam("email") String email) {

    if (securityContext.getUserPrincipal().getName().equals(email)
 ||
            securityContext.isUserInRole("ADMIN")) {
        DatastoreService datastore = DatastoreServiceFactory
                .getDatastoreService();

        Filter emailFilter = new FilterPredicate(PROP_EMAIL,
                FilterOperator.EQUAL, email);

        Query query = new Query(USER_KIND).setFilter(emailFilter);

        Entity userEntity = datastore.prepare(query).asSingleEntit
```

```
y();

        if (userEntity != null) {
            datastore.delete(userEntity.getKey());

            User user = entityToUser(userEntity);

            return user;
        } else {
            throw new WebApplicationException(Status.NOT_FOUND);

        }
    } else {
        throw new WebApplicationException(Status.FORBIDDEN);
    }
}
```

Repare que o teste para se certificar de que o usuário é `ADMIN` , ou é o próprio que será apagado, é exatamente o mesmo usado nas outras operações.

Atualizando o valor do registro do GCM

Por último, o **Requisito 7**, que se refere a uma operação para atualizar o valor do registro do usuário no GCM e a data que isso foi feito. Esse registro será mais bem explicado no capítulo *Enviando mensagens com o Google Cloud Messaging*, mas por enquanto, você só precisar fazer a operação e salvar o que vier, como mostra o código a seguir:

```
@PUT
@Produces(MediaType.APPLICATION_JSON)
@Path("/update_gcm_reg_id/{gcmRegId}")
@RolesAllowed({"USER"})
public User updateGCMRegId(@PathParam("gcmRegId") String gcmRegId)
{
    DatastoreService datastore = DatastoreServiceFactory
            .getDatastoreService();

    Filter emailFilter = new FilterPredicate(PROP_EMAIL,
            FilterOperator.EQUAL, securityContext.getUserPrincipal
().getName());

    Query query = new Query(USER_KIND).setFilter(emailFilter);
```

```
    Entity userEntity = datastore.prepare(query).asSingleEntity();


    if (userEntity != null) {

        userEntity.setProperty(PROP_GCM_REG_ID, gcmRegId);
        userEntity.setProperty(PROP_LAST_GCM_REGISTER, Calendar.ge
tInstance().getTime());

        User user = entityToUser(userEntity);

        datastore.put(userEntity);

        return user;
    } else {
        throw new WebApplicationException(Status.NOT_FOUND);

    }
}
```

Repare que não é necessário passar o endereço de e-mail do usuário que será alterado, pois como somente ele poderá acessar a operação, ele será localizado pelas informações contidas em `securityContext` .

Depois que o usuário é localizado, as propriedades `gcmRegId` e `lastGCMRegister` são atualizadas e salvas de volta no Datastore.

9.3 CONCLUSÃO

Neste capítulo, você construiu o serviço de usuários, além de ter incrementado o mecanismo de autenticação com novas funcionalidades como permissões baseadas em papéis. Você também aprendeu sobre como é possível fazer validação do modelo de dados utilizado nas operações, algo que pode economizar muito tempo e código, apenas usando anotações.

No próximo capítulo, você aprenderá sobre uma funcionalidade muito interessante do Google App Engine: enviar mensagens para

dispositivos móveis por meio de uma aplicação desenvolvida para o GAE. Isso equivale ao mecanismo conhecido como *Push Notification*, que auxilia muito os desenvolvedores do lado da aplicação móvel, como também os do lado da aplicação de back-end, a enviar notificações a usuários desses dispositivos.

CAPÍTULO 10

ENVIANDO MENSAGENS COM O GOOGLE CLOUD MESSAGING

Se você é um usuário comum de aplicativos de smartphone (Android, iOS, Windows Phone e outros), com certeza já deve ter utilizado alguma função de tais aplicativos, que permitia que você recebesse mensagens e notificações de algum servidor, como de e-mail, comunicador ou rede social.

Você já parou para se perguntar como isso realmente funciona? Será que esse aplicativo fica a todo instante consultando o servidor para saber se há novas mensagens para ele? Será que quem desenvolveu o sistema teve de manter uma conexão aberta o tempo todo para receber essas notificações?

Com certeza algum desses mecanismos citados foi usado, mas ficar consultando o servidor a cada 5 segundos, por exemplo, traria uma quantidade de requisições muito grande para ele, visto que pode haver milhares de clientes conectados, querendo saber se há alguma mensagem nova.

Utilizando o Google Cloud Messaging, é possível resolver todas essas questões de uma maneira bem simples, tanto para quem desenvolve o lado servidor do sistema quanto para quem desenvolve o aplicativo móvel.

10.1 O QUE É GOOGLE CLOUD MESSAGING

O Google Cloud Messaging (GCM) é um **serviço gratuito** do Google App Engine, que permite que você envie e receba mensagens entre aplicações servidoras e clientes. Isso quer dizer que você pode desenvolver uma aplicação no GAE para enviar mensagens para aplicativos móveis, no Android por exemplo, sem a preocupação de saber onde ele está localizado ou se está conectado à internet ou não.

Do lado da aplicação Android, você não precisa se preocupar em desenvolver códigos para ficar consultando o servidor a todo o momento e nem se preocupar em estabelecer uma conexão com ele.

Visão geral da arquitetura do GCM

A arquitetura completa do GCM inclui a aplicação responsável por gerar as mensagens que se conecta no GCM Connection Server, que é o responsável por gerenciar todo o mecanismo filas e de envio das mensagens para os aplicativos móveis e cuidar de seus registros, para que possam ser localizados. Veja a figura a seguir que ilustra tal arquitetura:

Figura 10.1: Arquitetura do GCM. Fonte: https://developers.google.com/cloud-messaging/gcm

É possível interagir com o GCM Connection Server de duas formas: conexão HTTP e XMPP. Este livro trata exclusivamente do

mecanismo de comunicação HTTP, por ser mais simples e fácil de se implementar. Para maiores informações sobre o mecanismo utilizando XMPP, consulte https://developers.google.com/cloud-messaging/ccs.

Utilizando o mecanismo de conexão HTTP, é possível desenvolver uma aplicação para enviar mensagens para o GCM Connection Server em vários formatos, como o JSON por exemplo, limitadas em ate 4 KB de tamanho.

Na arquitetura do GCM, o Connection Server é responsável por:

- Oferecer os mecanismos para que as aplicações móveis se registrem nele por meio de uma identificação única da aplicação servidora que deseja enviar mensagens;
- Receber as requisições das aplicações para envio de mensagens aos aplicativos móveis;
- Cuidar das filas de mensagens a serem entregues;
- Cuidar da entrega das mensagens aos aplicativos móveis;
- Garantir a tentativa de entrega das mensagens, mesmo que os aplicativos móveis não estejam online.

Com isso, sua aplicação que envia as mensagens não precisa se preocupar com esses pontos, bastando simplesmente enviar a mensagem para o destinatário desejado. Daí por diante, o GCM Connection Server faz todo o trabalho.

É importante de ressaltar que, utilizando o mecanismo de envio com HTTP, a aplicação que envia a mensagem não recebe uma confirmação da entrega de tal mensagem.

Conceitos importantes

O mecanismo do GCM possui alguns termos importantes que

devem ser entendidos para que se possa trabalhar com ele, principalmente para os desenvolvedores das aplicações servidora e móvel:

- **Sender ID**: é a identificação da aplicação responsável por enviar as mensagens ao GCM Connection Server. Em uma aplicação do GAE, isso equivale ao `Project Number`, que pode ser obtido na aba `Project Information` no menu de configurações do console do GAE, localizado no canto superior direito da tela.
- **API Key**: é a chave que garante o acesso à aplicação que deseja enviar mensagens ao GCM Connection Server. Isso será visto com mais detalhes na seção *Obtendo a API Key* deste capítulo.
- **Registration ID**: é a identificação da aplicação, quando ela se registra no GCM para receber as notificações.

PLATAFORMAS CLIENTE SUPORTADAS PELO GCM

É possível desenvolver aplicações para as plataformas móveis Android e iOS, para se registrarem e receberem notificações de uma aplicação servidora. Também é possível desenvolver uma aplicação ou extensão para o Google Chrome.

10.2 CONFIGURANDO O PROJETO NO GAE PARA UTILIZAR O GCM

Para que a aplicação servidora possa se conectar no GCM Connection Server e possa enviar mensagens a dispositivos móveis com Android e iOS, é necessário apenas adicionar algumas bibliotecas.

A primeira biblioteca é chamada `GCM Server`, responsável por toda a parte de conexão e comunicação com o GCM Connection Server. Para adicionar essa dependência ao projeto `exemplo1`, abra o arquivo `pom.xml` e coloque o seguinte trecho na seção dependências:

```
<dependency>
    <groupId>com.ganyo</groupId>
    <artifactId>gcm-server</artifactId>
    <version>1.0.2</version>
</dependency>
```

A segunda biblioteca é opcional e só deve ser adicionada caso você queira enviar mensagens no formato JSON para seus clientes. Ela é a biblioteca GSon do Google, para formatação de dados em JSON. Para isso, adicione a dependência a seguir no arquivo `pom.xml`:

```
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.5</version>
</dependency>
```

E é só isso, em termos de dependências, para que o projeto `exemplo1` fique apto a trabalhar com o GCM.

10.3 OBTENDO A API KEY

Para que a sua aplicação do GAE possa enviar mensagens pelo GCM, é necessário antes dar permissão a ela pelo console do GAE. Para isso, acesse o menu principal no canto superior esquerdo, e escolha a opção `API Manager -> Overview`, como mostra a figuar a seguir:

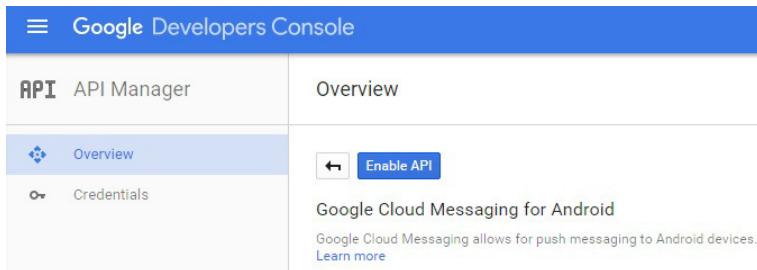



Figura 10.2: Habilitando o GCM

Clique no botão `Enable API` para habilitar a aplicação `exemplo1` a acessar a API do GCM para enviar mensagens a dispositivos móveis.

Nessa mesma tela, vai aparecer uma mensagem dizendo que essa API está habilitada, mas que não há nenhuma credencial cadastrada, e que por isso você ainda não pode usar. Clique no botão `Go to Credentials` para ir para a tela e criá-la, como mostra a figura a seguir:

Credentials

Add credentials to your project

1 Find out what kind of credentials you need

We'll help you set up the correct credentials
If you wish you can skip this step and create an API key, client ID, or service account

Which API are you using?

Determines what kind of credentials you need.

Google Cloud Messaging for Android

Where will you be calling the API from?

Determines which settings you'll need to configure.

Choose...

What credentials do I need?

Figura 10.3: Criando credencial para o GCM

Nessa tela, clique no link `API Key`, como está realçado na figura. Isso fará com que apareça uma caixa de diálogo perguntando qual o tipo de chave você deseja criar. Para esse caso, que é uma aplicação do GAE que vai utilizar essa credencial para enviar mensagens ao GCM, é necessário escolher a opção `Server Key`.

Figura 10.4: Server API Key

Na tela de criação da chave, digite um nome para ela, para que você possa identificá-la depois. Se for necessário, por questões de segurança, você pode definir um endereço ou uma faixa de IPs que estarão habilitados a utilizarem essa chave. Se nenhum endereço for colocado, significa que a chave poderá ser acessada de qualquer lugar da internet. Por enquanto, deixe esse campo vazio para que você possa testar a aplicação de qualquer lugar, seja hospedada no GAE ou na sua máquina desenvolvimento.

Figura 10.5: Criando Server API Key

Pronto! Agora você pode ver a sua chave criada:

Figura 10.6: Listando as API Key criadas

Esse o valor deverá ser usado dentro da aplicação Java para acessar o GCM e enviar mensagens aos dispositivos móveis.

10.4 O APLICATIVO MÓVEL PARA ANDROID

No repositório de código do livro (https://github.com/siecola/GAEBook), você pode encontrar uma aplicação Android de exemplo, que se registra para receber mensagens pelo GCM, através do `Sender ID` da aplicação hospedada no GAE. Depois de registrada, a aplicação espera por mensagens recebidas pelo GCM.

Quando uma mensagem é recebida, uma notificação aparece na barra de notificações do dispositivo. A aplicação foi projetada para receber qualquer notificação, em qualquer formato, exibindo-a exatamente como ela veio.

Se o usuário clicar na notificação de mensagem recebida, ela será exibida na tela principal da aplicação, mostrando todo o *payload* da mensagem.

Veja a figura a seguir da tela da aplicação Android de exemplo:

Figura 10.7: Aplicação Android de exemplo

Para se registrar na sua aplicação do GAE, estando com o dispositivo conectado na internet, basta digitar o `Sender ID` obtido no console do GAE, como explicado na seção *Conceitos*

importantes deste capítulo. Após isso, clique no botão `Register`, que a aplicação tentará se registrar e obter um `Registration ID`.

O valor obtido no campo `Registration ID` é o que deve ser utilizado quando a aplicação `exemplo1` desejar enviar uma mensagem para esse dispositivo. Esse campo permite que seu valor seja copiado para a área de transferência do Android. Esse mesmo valor é o que deve ser salvo na propriedade `gcmRegId` do tipo `User` no Datastore, criado no capítulo `Adicionando o serviço de usuários`.

Em uma aplicação real no Android, esse processo de registro no GCM seria feito em *background* e, em seguida, ela deveria acessar a operação `/update_gcm_reg_id/{gcmRegId}` do serviço de gerenciamento de usuários, atualizando o valor do `Registrarion ID` obtido. Isso não está sendo feito por essa aplicação de exemplo do Android. Por isso, você deve copiar o valor do campo `Registration ID` e atualizá-lo na operação citada do serviço de usuários, como mostra a figura a seguir:

Target

Target

Request URI

http://localhost:8888/api/users/update_gcm_reg_id/APA91bEabJceGjRvoJBoi

Universal Resource Identifier. ex: https://www.sample.com:9000

Request Method

PUT

The desired action to be performed on the identified resource.

Request Timeout

60 seconds

Timeout in seconds before aborting.

Figura 10.8: Atualizando o Registration ID

A figura a seguir mostra como a aplicação Android fica depois de registrar-se no GCM:

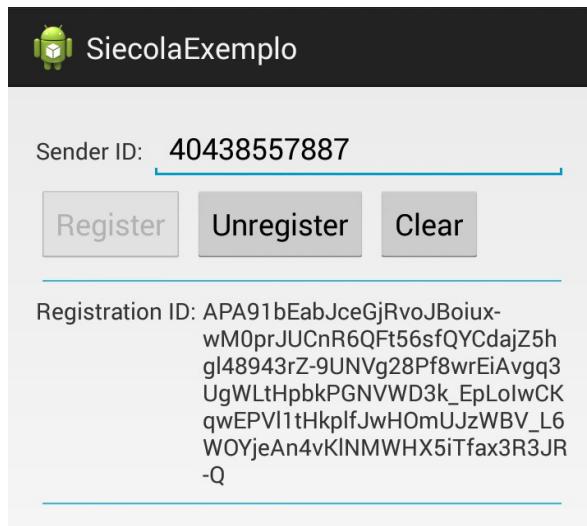



Figura 10.9: Aplicação registrada no GCM

Quando uma notificação for enviada a esse dispositivo através de seu `Registration ID`, pela aplicação `exemplo1` que está no GAE, o GCM Connection Server encaminhará essa mensagem pela Internet até chegar na aplicação exemplo do Android. Ele, por sua vez, vai exibir uma notificação na barra de notificações, e quando o usuário clicar nela, a mensagem será exibida na tela da aplicação, como mostra a figura a seguir:

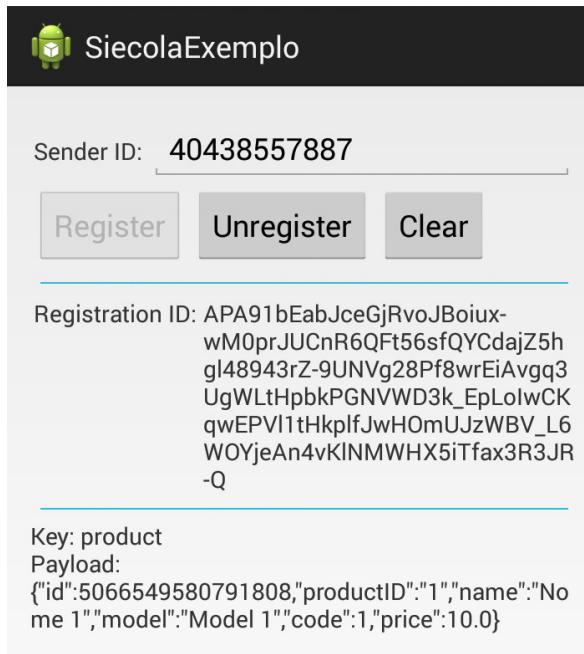



Figura 10.10: Mensagem recebida

Repare que a mensagem está sendo exibida em formato JSON, exatamente como foi recebida do GCM. Isso foi feito apenas para fins didáticos, para que você possa ver o *payload* que foi recebido.

A seção seguinte vai detalhar o que deve ser feito, na aplicação do `exemplo1` do GAE, para enviar mensagens a dispositivos móveis por meio do GCM.

10.5 ENVIANDO MENSAGENS A APLICATIVOS MÓVEIS COM O GCM

Para utilizar o GCM Connection Server e enviar mensagens a dispositivos móveis, que foram previamente registrados, são necessárias duas informações:

- A **API Key**, obtida através dos passos descritos na seção

Obtendo a API Key. Com essa chave, a aplicação servidora pode conectar-se no GCM Connection Server e enviar mensagens a dispositivos móveis;
- O **Registrarion ID** do dispositivo que deve receber a mensagem da aplicação servidora. Na aplicação `exemplo1` , essa informação está atrelada a cada usuário por meio da propriedade `gcmRegId` do tipo `User` , armazenada no Datastore. Quem deve fornecer esse valor é a aplicação móvel que se registrar no GCM. Cada dispositivo registrado no GDM, através do `Sender ID` da aplicação, possui um valor de registro diferente.

De posse desses dois dados, é possível enviar uma mensagem pelo GCM a qualquer dispositivo móvel registrado, executando os seguintes comandos de uma aplicação do GAE:

```
Sender sender = new Sender("API_KEY");
Message message = new Message.Builder().addData("IDENTIFICADOR_DA_
MENSAGEM", "MENSAGEM").build();
Result result = sender.send(message, "REGISTRATION_ID", 5);
```

Nesse trecho de código, o objeto do tipo `Sender` deve receber a **API KEY** obtida para a aplicação que será hospedada no GAE. O primeiro parâmetro de `Message.Builder().addData()` é uma identificação da mensagem, ou seja, um texto simples para dizer a aplicação móvel, por exemplo, qual é o tipo da mensagem. O segundo parâmetro é a mensagem em si, em formato texto, que pode ser um JSON.

No método que envia realmente a mensagem, ou seja, `sender.send()` , o primeiro parâmetro é a mensagem criada na linha anterior. O segundo é o **Registration ID** que a aplicação Android obteve ao se registrar no GCM, que no caso da aplicação `exemplo1` , é armazenada no Datastore pela operação `/update_gcm_reg_id/{gcmRegId}` do serviço de gerenciamento

de usuários. O terceiro parâmetro é o número de tentativas que o GCM deve fazer, caso não consiga enviar a mensagem na primeira vez.

> **ENVIANDO MENSAGENS DA MÁQUINA DE DESENVOLVIMENTO**
>
> É possível enviar mensagens através do GCM mesmo quando a aplicação servidora está em execução na máquina de desenvolvimento.

Para ilustrar melhor todo o mecanismo, crie um novo serviço, responsável por fazer esse trabalho na aplicação `exemplo1`. Para isso, crie uma nova classe chamada `MessageManager` no pacote `com.siecola.exemplo1.services`:

```
@Path("/message")
public class MessageManager {

    private static final Logger log = Logger.getLogger("MessageMan
ager");

    @Context
    SecurityContext securityContext;
}
```

Esse novo serviço será acessado pela URL `api/message`. Para tornar o exemplo mais interessante, será enviado como mensagem pelo GCM aos aplicativos móveis um dos produtos cadastrados no Datastore, usando o formato JSON.

Dentro desse serviço, crie uma operação para efetivamente enviar as informações do produto pelo GCM. Essa operação deve receber dois parâmetros: o código do produto a ser enviado e o endereço do e-mail do usuário. Através do endereço de e-mail, será possível localizar o usuário e o valor do registro de sua aplicação

móvel no GCM.

```
@POST
@Path("/sendproduct/{product_code}/{email}")
@RolesAllowed({"ADMIN"})
public String sendProduct (@PathParam("product_code") int productC
ode, @PathParam("email") String email) {

}
```

Essa operação deverá ser acessada pelo método `HTTP POST` por meio da URL `/sendproduct/{product_code}/{email}` .

Para tornar o exemplo um pouco mais simples, o produto e o usuário serão localizados no Datastore por meio de métodos privados criados na classe `MessageManager` , da mesma forma como já foi mostrado em capítulos anteriores:

```
private User findUser(String email) {
    DatastoreService datastore = DatastoreServiceFactory
            .getDatastoreService();

    Filter emailFilter = new FilterPredicate(UserManager.PROP_EMAI
L, FilterOperator.EQUAL, email);

    Query query = new Query(UserManager.USER_KIND).setFilter(email
Filter);
    Entity userEntity = datastore.prepare(query).asSingleEntity();

    if (userEntity != null) {
        return UserManager.entityToUser(userEntity);
    } else {
        return null;
    }
}

private Product findProduct (int code) {
    DatastoreService datastore = DatastoreServiceFactory
            .getDatastoreService();

    Filter codeFilter = new FilterPredicate("Code", FilterOperator
.EQUAL, code);

    Query query = new Query("Products").setFilter(codeFilter);
    Entity productEntity = datastore.prepare(query).asSingleEntity
();
```

```
    if (productEntity != null) {
        return ProductManager.entityToProduct(productEntity);
    } else {
        return null;
    }
}
```

Com isso, no início do método `sendProduct` criado para implementar a operação de enviar um produto, pode-se chamar esses dois métodos para localizar o produto e o usuário, utilizando os parâmetros recebidos da operação:

```
Product product;
if ((product = findProduct(productCode)) == null) {
    throw new WebApplicationException(Status.NOT_FOUND);
}

User user;
if ((user = findUser(email)) == null) {
    throw new WebApplicationException(Status.BAD_REQUEST);
}
```

Agora é só implementar a lógica para enviar a mensagem pelo GCM, da mesma forma como foi explicado no início desta seção, só que utilizando os dados retirados do Datastore. Além disso, colocando um pouco mais de tratamento, caso algo dê errado, como por exemplo, o usuário não estar registrado no GCM:

```
Sender sender = new Sender("API_KEY");
Gson gson = new Gson();
Message message = new Message.Builder().addData("product", gson.to
Json(product)).build();
Result result;

try {
    result = sender.send(message, user.getGcmRegId(), 5);
    if (result.getMessageId() != null) {
        String canonicalRegId = result.getCanonicalRegistrationId(
);
        if (canonicalRegId != null) {
            log.severe("Usuário [" + user.getEmail() + "] com mais
 de um registro");
        }
    } else {
```

```
        String error = result.getErrorCodeName();
        log.severe("Usuário [" + user.getEmail() + "] não registra
do");
        log.severe(error);
        throw new WebApplicationException(Status.NOT_FOUND);

    }
} catch (IOException e) {
    throw new WebApplicationException(Status.INTERNAL_SERVER_ERROR
);
}
return "Produto " + product.getName() + " enviado com sucesso para
 usuário " + user.getEmail();
```

Na linha `Sender sender = new Sender("API_KEY");` , você deve substituir a string `API_KEY` pela chave obtida para a sua aplicação `exemplo1` , como detalha os passos da seção *Obtendo a API Key*.

O teste logo abaixo da linha `result = sender.send(message, user.getGcmRegId(), 5);` serve para verificar se o usuário está realmente registrado no GCM com o registro armazenado no Datastore. Dessa forma, se ele não estiver, a operação retorna com o código `HTTP 404 Not Found` .

Se tudo der certo e o GCM receber a mensagem, a operação retorna com o código `HTTP 200 OK` , informando que a mensagem foi entregue ao GCM.

ENVIO DA MENSAGEM PELO GCM

Como foi dito no início deste capítulo, quando se utiliza o mecanismo HTTP, a mensagem é entregue ao GCM, mas não há confirmação de que ela realmente foi entregue ao dispositivo móvel.

A aplicação que envia a mensagem não recebe uma resposta, positiva ou negativa, que a mensagem foi entregue ao dispositivo móvel. Isso é uma característica quando se usa o tipo de conexão HTTP com o GCM Connection Server.

Para testar o funcionamento desse serviço, basta usar o REST Console, passando um código de um produto armazenado no Datastore e o e-mail do usuário que você tenha cadastrado o `Registration ID`, como mostra a figura a seguir:

Figura 10.11: Testando o serviço

O mecanismo do GCM é muito interessante e, ao mesmo tempo, fácil de ser utilizado. Como você pode observar, toda a tarefa pesada de enfileirar as mensagens e tentar entregá-las a cada dispositivo móvel, fica a cargo do GCM Connection Server. A aplicação deve simplesmente conectar-se nele e enviar a mensagem.

10.6 CONCLUSÃO

Parabéns! Você construiu um serviço REST que aciona o mecanismo do GCM para enviar mensagens a dispositivos móveis. Esse serviço pode ser usado por outras aplicações, que não estão rodando no GAE, para solicitar que tal trabalho seja feito.

No próximo capítulo, você aprenderá mais uma funcionalidade interessante que a plataforma do Google App Engine oferece: agendamento de tarefas. Esse é um recurso muito interessante quando você deseja executar alguma tarefa, em um intervalo de tempo pré-determinado.

CAPÍTULO 11

AGENDANDO TAREFAS NO GAE

Em aplicações Web, a necessidade de agendar tarefas para serem executadas de forma automática é muito comum. Por exemplo, para enviar e-mails durante a faixa de horário com menor tráfego de rede, executar limpeza de banco de dados ou arquivos de log etc.

11.1 COMO FUNCIONAM AS TAREFAS AGENDADAS NO GAE

Com o Google App Engine, é possível programar uma tarefa para ser executada automaticamente pela plataforma. Essa tarefa, então, vai acessar um serviço da sua aplicação, utilizando o método `HTTP GET` . Dessa forma, basta criar um serviço e uma operação, especificamente para ser chamado pela tarefa agendada. Dentro do método, que tratará a requisição a essa operação, você pode realizar a atividade programada.

A operação que é acessada pela tarefa agendada é igual a qualquer outra, com as mesmas regras, como por exemplo, o tempo execução e resposta. Além disso, para que o GAE considerar que a tarefa foi executada com sucesso pela sua aplicação, a operação deve retornar com o código HTTP com os valores entre 200 e 299, inclusive.

No GAE, as aplicações no regime gratuito da plataforma podem

ter até 20 tarefas agendadas, enquanto as que estão com o mecanismo de cobrança ativado podem ter até 100.

> **TAREFAS AGENDADAS NO AMBIENTE DE DESENVOLVIMENTO**
>
> As tarefas agendas não funcionam no ambiente de desenvolvimento, então você deve obrigatoriamente publicar a aplicação para vê-las funcionando.

11.2 CRIANDO O NOVO SERVIÇO AGENDADO

Para exemplificar o mecanismo de tarefas agendadas, crie um novo serviço com uma única operação, para imprimir uma mensagem no log da aplicação.

> Para facilitar o trabalho e deixar o foco somente no agendamento da tarefa, o novo serviço não deverá passar pelo mecanismo de autenticação.

Para que o serviço não faça parte do mecanismo de autenticação, crie um novo pacote na aplicação com o nome de `com.siecola.exemplo1.cronservices` . Nesse novo pacote, crie a classe que vai implementar o serviço e a operação que será chamada pela tarefa agendada:

```
@Path("cron1")
public class CronService {

    private static final Logger log = Logger.getLogger("CronServic
e");

    @GET
```

```
    @Produces("application/json")
    @Path("testcron")
    public void testCron() {
        log.severe("Cron message --- " + Calendar.getInstance().ge
tTime());
    }
}
```

Como você pode ver, ele é um método comum para implementar uma operação normal, sem nada de novo. A sua URL de acesso, como foi configurado pelas anotações, será: `cron/cron1/testcron` , sendo que a primeira parte da URL, ou seja, `cron` , será configurada no arquivo `web.xml` . O que essa operação faz é apenas exibir uma mensagem no log da aplicação, quando ela for chamada pela tarefa agendada.

Como esse serviço está fora do pacote dos demais serviços, é necessário adicionar uma nova configuração no arquivo `web.xml` , fazendo com que o novo pacote e a nova classe sejam enxergados pelo Jersey como serviços, além de dizer que eles não devem passar pelo mecanismo de autenticação. Veja como deve ficar a nova configuração nesse arquivo:

```
<servlet>
    <servlet-name>CronServices</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</
servlet-class>
    <init-param>
        <param-name>jersey.config.server.provider.packages</param-
name>
        <param-value>com.siecola.exemplo1.cronservices</param-value
>
    </init-param>
    <init-param>
        <param-name>org.glassfish.jersey.api.json.POJOMappingFeatu
re</param-name>
        <param-value>true</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>CronServices</servlet-name>
    <url-pattern>/cron/*</url-pattern>
```

```
</servlet-mapping>
```

Repare que a configuração para o novo servlet não inclui o parâmetro para que as requisições a ele passem pelo filtro, que faz o mecanismo de autenticação. Veja também que a URL configurada é `/cron/*` .

11.3 CONFIGURANDO A TAREFA

Para configurar as tarefas agendadas da aplicação, é necessário criar o arquivo de nome `cron.xml` , dentro da pasta `src/main/webapp/WEB-INF/` . Nele, é possível configurar mais de uma tarefa, com agendamentos e URLs diferentes.

Veja no exemplo a seguir como deve ficar o agendamento de uma tarefa a ser executada a cada dois minutos para acessar o serviço que foi criado na seção anterior:

```
<?xml version="1.0" encoding="UTF-8"?>
<cronentries>
  <cron>
    <url>/cron/cron1/testcron</url>
    <description>Mensagem do Cron a cada 2 minutos</description>
    <schedule>every 2 minutes</schedule>
  </cron>
</cronentries>
```

O campo `url` define o endereço da operação a ser chamada quando a tarefa tiver de ser executada. No campo `description` , você pode colocar um texto descritivo sobre a tarefa. Essa informação aparecerá no console do GAE, na seção de gerenciamento das tarefas agendadas. E finalmente, no campo `schedule` é onde você deve definir a periodicidade de execução da tarefa.

Formato do agendamento

As tarefas podem ser agendadas para executarem de diversas

formas:

- Com uma periodicidade de tempo (minutos, horas ou dias), por meio do parâmetro `every` ;
- Com uma periodicidade de tempo (minutos, horas ou dias) e dentro de um intervalo de horas, combinando o parâmetro `every` com o `from` e `to` ;
- A cada hora de todos os dias ou de dias específicos;
- Dentro de intervalos de tempo, com uma periodicidade, todos os dias ou em dias específicos, por exemplo, utilizando a expressão `every monday 09:00` .

Para mais exemplos sobre o formato a ser usado no campo `schedule` do arquivo `cron.xml` , para configurar a periodicidade da execução da tarefa, consulte a documentação do Google em https://cloud.google.com/appengine/docs/java/config/cron#Java_appengine_web_xml_The_schedule_format.

11.4 ACOMPANHANDO A EXECUÇÃO DO CONSOLE DO GAE

Para visualizar a tarefa em execução no console do GAE, é necessário que você publique sua aplicação antes. Depois de publicar, vá ao console do GAE, dentro do menu `App Engine` da sua aplicação `exemplo` , e acesse a opção `Task queues` . Na aba `Cron Jobs` , é onde ficam as tarefas agendas criadas da forma como foi apresentado neste capítulo.

Nessa aba, você verá as estatísticas de execução da tarefa que foi agendada, como o endereço da operação chamada, sua descrição, a frequência de execução, a última vez que ela rodou e se foi executada com sucesso ou não.

Figura 11.1: Monitorando tarefas agendadas

Como a tarefa criada simplesmente exibe uma mensagem no log, vá até a seção que exibe as mensagens no console do GAE para certificar-se de que ela realmente está escrevendo a mensagem a cada 2 minutos, como foi programada.

Figura 11.2: Mensagem da tarefa agendada

11.5 CONCLUSÃO

Neste capítulo, você aprendeu mais uma funcionalidade da plataforma do Google App Engine para agendar tarefas para serem executadas de forma automática por ele, de dentro da sua aplicação, sem grandes esforços, apenas configurando um arquivo XML.

No próximo capítulo, você utilizará outra funcionalidade interessante do GAE, que o *memory cache*, usado para manter dados na memória, sem utilizar o Datastore.

CAPÍTULO 12

UTILIZANDO MEMORY CACHE

O mecanismo de autenticação de usuários da aplicação `exemplo1` atualiza a data/hora na propriedade `lastLogin` no tipo `Users` do Datastore toda a vez que ele acessa o sistema. Essa informação pode ser usada para saber, por exemplo, qual foi a última vez que ele acessou o sistema e também para saber quais são os usuários que há muito não utilizam a aplicação.

Como você deve perceber, essa informação é muito útil para fins estatísticos, pois é importante saber se os usuários da sua aplicação são assíduos. Porém, é bem provável que você não necessite de uma informação tão precisa, a ponto de indicar exatamente a data/hora do último acesso, com minutos e segundos de precisão. Talvez bastasse que tivesse a precisão de minutos ou horas.

Então, como fazer para gerenciar todas as requisições de autenticação e, ao mesmo tempo, atualizar a data/hora de último acesso de apenas algumas requisições, em intervalos de tempo de alguns minutos, para fins estatísticos? Isso pode ser feito com um serviço muito interessante da plataforma do Google App Engine, chamado **MemCache**.

12.1 O QUE É MEMCACHE

O MemCache é uma funcionalidade do GAE que permite que a

aplicação em Java possa salvar em memória, informações sob a forma de **conjuntos chave/valor**, sem persistência desses dados, como é feito no Datastore.

Como esse mecanismo faz o seu trabalho em memória e sem acesso a disco, o tempo de acesso para leitura e escrita é muito mais rápido que no Datastore. Por isso, ele pode ser utilizado para armazenar valores temporários, sem a necessidade de persisti-los.

O MemCache pode ser utilizado em situações como:

- Armazenamento de preferências de usuários;
- Evitar consultas e/ou escritas sucessivas ao Datastore, de informações que certamente não se alteraram a todo instante;
- Armazenar informações temporárias, dentro do contexto de algumas requisições em um período de tempo conhecido.

É importante ressaltar que as informações guardadas no MemCache, caso não seja explicitamente declarado um tempo para sua expiração, ficaram armazenadas o maior tempo possível, dependendo da utilização e do espaço disponível, que são gerenciados pela plataforma do GAE. No próximo capítulo, será apresentada uma forma de configurar esse tempo, de acordo com a necessidade da aplicação.

Os dados guardados no MemCache são de comum acesso por toda a aplicação, tornando o seu uso muito simples, pois um dado pode ser salvo por uma operação de um serviço e utilizado por outro completamente diferente, devendo apenas ser conhecida a chave de seu valor. Além disso, na modalidade gratuita do serviço, a infraestrutura de armazenamento dos dados é compartilhada com as demais aplicações do GAE, sendo da mesma aplicação ou não. Para obter uma infraestrutura exclusiva para a sua aplicação, além

de maior capacidade de armazenamento, é necessário optar para a opção onde o serviço é pago.

Limites de utilização do MemCache

O serviço MemCache do GAE possui alguns limites de utilização, tais como:

- O tamanho máximo de um valor a ser armazenado é de 1 MByte, menos o tamanho da chave e do *overhead* gerado pela implementação de acesso ao MemCache, como o JCache;
- O tamanho da chave não pode ser maior do que 250 bytes;
- O valor da chave não pode ser nulo.

12.2 UTILIZANDO JCACHE

O MemCache pode ser usado pela sua API nativa, acessando diretamente o serviço do GAE. Porém, o SDK do App Engine possui uma implementação do JCache (JSR 107) para acessá-lo, o que torna o trabalho muito mais fácil e prático.

Para utilizar o MemCache com o JCache, basta usar a interface `javax.cache.Cache`. Dessa forma, uma instância do objeto do tipo `Cache` pode ser obtida por meio do `CacheFactory`, como pode ser visto no trecho de código a seguir:

```
try {
    CacheFactory cacheFactory = CacheManager.getInstance().getCach
eFactory();
    cache = cacheFactory.createCache(Collections.emptyMap());

} catch (CacheException e) {
    //Tratar exceção
}
```

Inserindo e buscando valores no MemCache

De posse da instância de `Cache`, para inserir um valor no MemCache, basta executar o método `put`:

```
cache.put("chave", "valor");
```

Lembrando de que o método `put` pode receber valores do tipo `Object` como parâmetros.

Para ler algum valor do MemCache, basta utilizar o método `get`:

```
String valor = (Date)cache.get("chave");
```

Se a chave não existir no MemCache, o objeto retornado será nulo.

Verificando se uma chave existe no MemCache

Para verificar se uma chave existe no Memcache, pode-se utilizar o método `containsKey`, que recebe como argumento a chave de pesquisa, retornando verdadeiro, caso encontre, e falso, caso contrário.

Removendo uma chave do MemCache

Para remover uma chave do MemCache, basta usar o método `remove`, que recebe como parâmetro o valor da chave a ser removida:

```
cache.remove("chave");
```

12.3 USANDO MEMCACHE NO MECANISMO DE AUTENTICAÇÃO

Nesta seção, será mostrado como utilizar o serviço MemCache

do GAE para melhorar o desempenho do mecanismo de autenticação da aplicação `exemplo1` , persistindo a informação do último login do usuário somente se ele acessou algum serviço dessa aplicação há mais de 30 segundos. Isso fará com que o número de acessos ao Datastore para realizar essa tarefa caia bastante, pois atualmente em toda requisição que é feita, essa informação é atualizada é persistida nele.

Para começar, desacople a persistência da informação do usuário no Datastore do método `checkCredentialsAndRoles` da classe `AuthFilter` , do pacote `com.siecola.exemplo1.authentication` da aplicação `exemplo1` , criando um novo método privado para realizar tal tarefa, que agora vai utilizar o MemCache. Para isso, retire o seguinte trecho desse método:

```
final User user = new User();

user.setEmail((String) userEntity.getProperty(UserManager.PROP_EMA
IL));
user.setPassword((String) userEntity.getProperty(UserManager.PROP_
PASSWORD));
user.setId(userEntity.getKey().getId());
user.setGcmRegId((String) userEntity.getProperty(UserManager.PROP_
GCM_REG_ID));
user.setLastLogin((Date) Calendar.getInstance().getTime());
user.setLastGCMRegister((Date) userEntity.getProperty(UserManager.
PROP_LAST_GCM_REGISTER));
user.setRole((String) userEntity.getProperty(UserManager.PROP_ROLE
));

userEntity.setProperty(UserManager.PROP_LAST_LOGIN, user.getLastLo
gin());
datastore.put(userEntity);
```

Substitua esse trecho pela chamada do novo método:

```
final User user = updateUserLogin(datastore, userEntity);
```

Agora crie o novo método, a princípio para realizar os mesmos passos que o trecho retirado fazia:

```
private User updateUserLogin (DatastoreService datastore, Entity u
serEntity) {
    User user = new User();

    user.setEmail((String) userEntity.getProperty(UserManager.PROP
_EMAIL));
    user.setPassword((String) userEntity.getProperty(UserManager.P
ROP_PASSWORD));
    user.setId(userEntity.getKey().getId());
    user.setGcmRegId((String) userEntity.getProperty(UserManager.P
ROP_GCM_REG_ID));
    user.setLastLogin((Date) Calendar.getInstance().getTime());
    user.setLastGCMRegister((Date) userEntity.getProperty(UserMana
ger.PROP_LAST_GCM_REGISTER));
    user.setRole((String) userEntity.getProperty(UserManager.PROP_
ROLE));

    userEntity.setProperty(UserManager.PROP_LAST_LOGIN, user.getLa
stLogin());
    datastore.put(userEntity);

    return user;
```

Nesse momento, a lógica de persistir a informação no Datastore apenas foi retirada para esse método exclusivo a isso.

A lógica proposta para utilizar o MemCache tem os seguintes requisitos principais:

- Salvar no MemCache o endereço de e-mail do usuário autenticado, juntamente com a data/hora que isso ocorreu;
- Durante o tratamento de uma requisição, caso essa informação ainda não exista no MemCache, ela deverá ser salva nele e no Datastore também;
- Se a informação já existir no MemCache e a data/hora da última atualização for menor do que 30 segundos em relação à nova requisição, nem o MemCache nem o Datastore deverão ser atualizados;
- Se a informação do MemCache for mais antiga do que 30 segundos, então tanto ele quanto o Datastore

deverão ter os valores do último login atualizados para o usuário em questão.

Como você pode perceber, o número de acessos ao Datastore para a atualização da informação de data/hora do último login de cada usuário reduzirá bastante. O tempo de 30 segundos escolhido para esse exemplo pode até ser maior, em uma aplicação real, dependendo da necessidade, reduzindo ainda mais os acessos ao Datastore para gravação de dados.

Para implementar os requisitos apresentados, comece alterando o método `updateUserLogin` , incluindo o código responsável por realizar a lógica proposta por eles:

```
User user = new User();
boolean canUseCache = true;
boolean saveOnCache = true;

String email = (String) userEntity.getProperty(UserManager.PROP_EM
AIL);

Cache cache;
try {
    CacheFactory cacheFactory = CacheManager.getInstance().getCach
eFactory();
    cache = cacheFactory.createCache(Collections.emptyMap());

    if (cache.containsKey(email)) {
        Date lastLogin = (Date)cache.get(email);
        if ((Calendar.getInstance().getTime().getTime() - lastLogi
n.getTime()) < 30000) {
            saveOnCache = false;
        }
    }

    if (saveOnCache == true) {
        cache.put(email, (Date)Calendar.getInstance().getTime());

        canUseCache = false;
    }
} catch (CacheException e) {
    canUseCache = false;
}
```

Repare que, depois que a instância do MemCache foi requisitada, o código verifica a existência de uma chave com o valor do e-mail do usuário autenticado, pelo método `cache.containsKey(email)` . Caso exista algum valor, a data lá armazenada é comparada com a atual do sistema. Se o intervalo for maior do que 30 segundos, a informação é atualizada no MemCache e posteriormente no Datastore, como pode ser visto no restante do código desse método:

```
if (canUseCache == false) {
    user.setEmail((String) userEntity.getProperty(UserManager.PROP
_EMAIL));
    user.setPassword((String) userEntity.getProperty(UserManager.P
ROP_PASSWORD));
    user.setId(userEntity.getKey().getId());
    user.setGcmRegId((String) userEntity.getProperty(UserManager.P
ROP_GCM_REG_ID));
    user.setLastLogin((Date) Calendar.getInstance().getTime());
    user.setLastGCMRegister((Date) userEntity.getProperty(UserMana
ger.PROP_LAST_GCM_REGISTER));
    user.setRole((String) userEntity.getProperty(UserManager.PROP_
ROLE));

    userEntity.setProperty(UserManager.PROP_LAST_LOGIN, user.getLa
stLogin());
    datastore.put(userEntity);
}

return user;
```

Perceba que o código que atualiza o Datastore só é executado caso a informação não esteja no MemCache ainda, ou se já se passaram 30 segundos desde a última atualização do MemCache para esse usuário. Isso reduz os acessos de escrita ao Datastore para cada usuário.

12.4 VISUALIZANDO O MEMCACHE DO CONSOLE DO GAE

O console do GAE possui uma seção específica para exibir as

informações que estão armazenadas no MemCache. Para acessá-las, basta ir ao menu principal da aplicação na seção `App Engine` e escolher a opção `MemCache` :

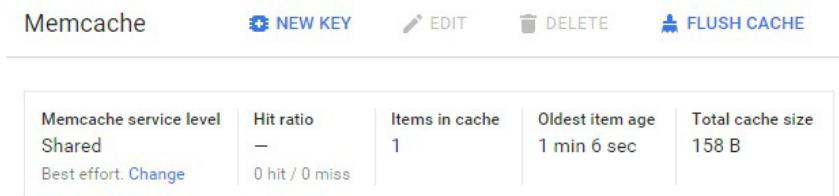



Figura 12.1: Estatísticas do MemCache

Nessa mesma tela, é possível localizar os dados armazenados no MemCache, localizando-os por meio da sua chave:

Figura 12.2: Dados do MemCache

MEMCACHE NO AMBIENTE DE DESENVOLVIMENTO

O MemCache também funciona no ambiente de desenvolvimento, mas o console de administração local não possui uma interface para exibir os dados contidos nele. Isso só está disponível na interface do console do GAE do ambiente de produção.

12.5 CONCLUSÃO

Neste capítulo, você melhorou o desempenho do mecanismo de autenticação de usuários, utilizando o serviço MemCache do GAE, mais uma funcionalidade muito interessante e útil da plataforma do Google.

No próximo capítulo, você aprenderá mais uma funcionalidade do MemCache, que é a configuração do tempo que a informação pode ficar armazenada no memory cache, antes de ser automaticamente apagada. Isso será muito útil para o gerenciamento de tokens, utilizado no mecanismo de autenticação OAuth, que será o foco principal do capítulo, onde será apresentada uma nova forma de autenticação de usuários.

Capítulo 13

PROTEGENDO SERVIÇOS COM OAUTH 2

A segurança em serviços REST deve ser muito bem tratada, pois como foi dito no início do capítulo *Protegendo serviços com HTTP Basic Authentication*, vários aspectos devem ser levados em consideração na escolha do melhor mecanismo a ser implementado no servidor que provê os serviços.

Um tipo de autenticação que vem se popularizando muito é o OAuth, principalmente por alguns pontos importantes:

- Facilidade de implementação, tanto pelo provedor quanto pelo cliente;
- Pouco consumo de recursos do lado do servidor;
- Um menor *overhead* de requisições para que o cliente consiga acesso autenticado aos recursos.

Esse último é muito importante, principalmente para dispositivos móveis, onde é necessário balancear o nível de segurança do serviço em questão e a quantidade de requisições que a aplicação faz, pois muitas vezes o acesso à internet tem custo.

Este capítulo apresenta o mecanismo de autenticação **OAuth 2 com Bearer Token**, para proteger os serviços hospedados no GAE. Você verá que muito do que já foi aprendido no capítulo *Protegendo serviços com HTTP Basic Authentication* será reaproveitado, como o

mecanismo de filtro de requisições, anotações nos métodos das classes dos serviços e como fazer para descobrir qual é o usuário autenticado, dentro desses métodos.

13.1 O QUE É OAUTH 2

Proteger serviços REST com o mecanismo OAuth 2 é uma escolha sensata para a maioria das aplicações, visto que traz um bom equilíbrio entre o nível de segurança oferecido por esse método, consumo de recursos do servidor e números de requisições do cliente.

Basicamente, o OAuth 2 com Bearer Token funciona com os seguintes elementos:

- **Token:** é uma chave de acesso que o cliente deve obter junto ao provedor de autenticação para ser utilizada em todas as requisições aos serviços;
- **Provedor de autenticação:** é o responsável por fornecer os tokens para os clientes que solicitarem. Também cuida da validade de cada token, para que eles existam somente durante um tempo determinado pelo provedor;
- **Mecanismo de autenticação:** verifica, em cada requisição do cliente, se há um token válido, e localiza o usuário que o solicitou no provedor de autenticação, para que a requisição possa prosseguir com um usuário autenticado associado a ela.

O provedor de autenticação só entrega o token a um cliente se ele fornecer as suas credenciais de acesso corretas, cadastras em um banco de dados ou em outro local apropriado.

Perceba que após o cliente solicitar o token ao provedor de

autenticação, ele não fornece mais as credenciais de acesso nas demais requisições aos serviços que deseja acessar. Ele utiliza somente o token para isso, até que ele expire e tenha de solicitar outro. Dessa forma, se alguém interceptar a comunicação e obter o token do cliente, ele não terá acesso às credenciais de acesso de seu usuário.

13.2 CRIANDO A APLICAÇÃO EXEMPLOOAUTH

Para demonstrar como funciona o mecanismo de autenticação de usuários usando OAuth com Bearer Token, crie uma nova aplicação chamada `exemplooauth` , com apenas o serviço de usuários, para tornar seu trabalho mais fácil e focado.

O projeto `exemplooauth` deve ser criado seguindo os passos descritos no capítulo *Desenvolvendo a primeira aplicação para o GAE*, na seção *Construindo o projeto básico*. Muita coisa será copiada do projeto `exemplo1` , para deixar a explicação mais focada no objetivo deste capítulo. Lembrando, você pode baixar o código de todos os projetos deste livro, no repositório https://github.com/siecola/GAEBook.

Adicionando as dependências ao projeto

Para auxiliar na geração dos tokens para os usuários e interpretar a requisição que o solicita, será utilizada a biblioteca Apache Oltu. Para adicionar as bibliotecas necessárias ao projeto, configure o arquivo `pom.xml` , adicionando as seguintes dependências do Jersey e do Apache Oltu:

- Dependências do Jersey

```
<dependency>
    <groupId>org.glassfish.jersey.containers</groupId>
```

```
    <artifactId>jersey-container-servlet</artifactId>
    <version>2.22.1</version>
</dependency>

<dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-json-jackson</artifactId>
    <version>2.22.1</version>
</dependency>

<dependency>
    <groupId>org.glassfish.jersey.ext</groupId>
    <artifactId>jersey-bean-validation</artifactId>
    <version>2.22.1</version>
</dependency>
```

- Dependências do Apache Oltu

```
<dependency>
    <groupId>org.apache.oltu.oauth2</groupId>
    <artifactId>org.apache.oltu.oauth2.authzserver</artifactId>
    <version>1.0.1</version>
</dependency>

<dependency>
    <groupId>org.apache.oltu.oauth2</groupId>
    <artifactId>org.apache.oltu.oauth2.resourceserver</artifactId>
    <version>1.0.1</version>
</dependency>
```

Configurando o arquivo web.xml

Não há nada de diferente na configuração do arquivo `web.xml` do projeto `exemplooauth` em relação ao `exemplo1` , pois a mesma técnica de filtros de requisições será usada. Também não é necessário adicionar nenhuma configuração extra nesse arquivo por conta do mecanismo de autenticação OAuth.

Veja como ele deve ficar com a configuração do pacote onde devem ficar as classes dos serviços, assim como o filtro de requisições:

```
<!--  Services -->
<servlet>
```

```
    <servlet-name>exemplooauth_services</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</
servlet-class>
    <init-param>
        <param-name>jersey.config.server.provider.packages</param-
name>
        <param-value>com.siecola.exemplooauth.services</param-value
>
    </init-param>
    <init-param>
        <param-name>jersey.config.server.provider.classnames</para
m-name>
        <param-value>com.siecola.exemplooauth.authentication.AuthF
ilter;org.glassfish.jersey.filter.LoggingFilter</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>exemplooauth_services</servlet-name>
    <url-pattern>/api/*</url-pattern>
</servlet-mapping>

<listener>
    <listener-class>com.siecola.exemplooauth.InitServletContextCla
ss</listener-class>
</listener>
```

13.3 CRIANDO O SERVIÇO DE USUÁRIOS

Como o projeto `exemplooauth` só terá o serviço de usuários, principalmente porque essa base será utilizada pelo mecanismo OAuth, você deve criá-lo da mesma forma como foi feito no projeto `exemplo1`.

Modelo User

O modelo de usuário deve ser a primeira coisa a ser criada. Ele será o mesmo usado no projeto `exemplo1`. Por isso, crie o pacote `com.siecola.exemplooauth.models`.

Em seguida, copie a classe `User` da aplicação `exemplo1` e faça as devidas alterações de nomes de pacotes para o novo projeto

`exemplooauth`.

Serviço de usuários

O serviço de usuários também será o mesmo que foi utilizado no projeto `exemplo1`, com as mesmas anotações e regras. Para isso, crie o pacote `com.siecola.exemplooauth.services`.

Depois, copie a classe `UserManager` da aplicação `exemplo1` e faça as devidas alterações de nomes de pacotes para o novo projeto `exemplooauth`.

Inicializando os dados de usuários

Para inicializar a base de usuários, crie a classe `InitServletContextClass`, que de acordo com as configurações feitas no arquivo `web.xml`, será chamada assim que a aplicação subir. Para isso, crie o pacote `com.siecola.exemplooauth`.

Por último, copie a classe `InitServletContextClass` e faça as devidas alterações de nomes de pacotes para o novo projeto `exemplooauth`.

13.4 FORNECENDO OS TOKENS DE AUTENTICAÇÃO

O provedor de autenticação, na verdade, será um serviço para fornecer os tokens aos clientes quando eles solicitarem. Obviamente, esse serviço deve estar acessível a qualquer usuário, mesmo que ele não esteja autenticado.

Para que o usuário receba o token, é necessário que ele esteja cadastrado no Datastore. Por isso, ele deverá fornecer suas credenciais corretas ao provedor de autenticação.

Parte do código do provedor de autenticação que será apresentado foi retirado da própria documentação e projetos de exemplo do Apache Oltu, que você pode encontrar em https://oltu.apache.org/.

Para começar, crie uma nova classe chamada `OAuthToken`. Ela implementará o serviço onde os clientes solicitarão o token de acesso, por meio da URL `api/token`.

```
@Path("/token")
public class OAuthToken {

}
```

Esse serviço terá apenas uma operação, que será onde os tokens poderão ser requisitados. Ela deverá ser acessada utilizando o método `HTTP POST`, recebendo no corpo da mensagem no formato `application/x-www-form-urlencoded` as credenciais de acesso do usuário que solicita o token. Além disso, a operação deverá responder com o token de acesso, no formato `application/json`. Para isso, crie o método `autorize`, como mostrado a seguir:

```
@POST
@Consumes("application/x-www-form-urlencoded")
@Produces("application/json")
@PermitAll
public Response authorize(@Context HttpServletRequest request, Mul
tivaluedMap<String, String> form)
        throws OAuthSystemException {

}
```

Como dito anteriormente, essa operação poderá ser acessada por qualquer usuário cadastrado, mesmo que ele não esteja autenticado ou com um token que recebeu previamente. Por isso, a anotação `@PermitAll` deve ser usada.

Para que seja possível acessar o corpo da requisição, que carrega as credenciais do cliente, é necessário criar uma classe que estende de `HttpServletRequestWrapper`, pois o Jersey por padrão não

passa a mensagem que recebeu no corpo da mensagem. Para isso, no mesmo arquivo onde a classe `OAuthToken` foi criada, crie a classe `OAuthRequestWrapper`, sobrescrevendo os métodos `getParameter` e `getParameterValues`:

```
class OAuthRequestWrapper extends HttpServletRequestWrapper {
    private MultivaluedMap<String, String> form;

    public OAuthRequestWrapper(HttpServletRequest request,
            MultivaluedMap<String, String> form) {
        super(request);
        this.form = form;
    }

    @Override
    public String getParameter(String name) {
        String value = super.getParameter(name);
        if (value == null) {
            value = form.getFirst(name);
        }
        return value;
    }

    @Override
    public String[] getParameterValues(String name) {
        String[] values = super.getParameterValues(name);
        if(values == null && form.get(name) != null){
            values = new String[form.get(name).size()];
            values = form.get(name).toArray(values);
        }
        return values;
    }
}
```

O formato das credenciais de acesso que o cliente deve enviar na requisição para pegar o token deve ter o seguinte formato:

```
grant_type=password&username=admin@siecola.com&password=Admin#7&cl
ient_secret=Admin#7&client_id=admin@siecola.com
```

Sendo que nesse exemplo, o login do usuário é `admin@siecola.com` e a senha é `Admin#7`. Os parâmetros `client_secret` e `client_id` são exigidos pelo interpretador do Apache Oltu e podem ser utilizados para identificar a aplicação

cliente, que está fazendo a requisição em nome de um usuário.

Para iniciar com a implementação do método `autorize`, comece realizando a interpretação da mensagem com as credenciais do usuário e também gerando o número que representará o token de acesso, como mostra o trecho a seguir:

```
try {
OAuthTokenRequest oauthRequest = new OAuthTokenRequest(new OAuthRe
questWrapper(request, form));
OAuthIssuer oauthIssuerImpl = new OAuthIssuerImpl(
        new MD5Generator());
```

Veja que o objeto `OAuthTokenRequest oauthRequest` é criado a partir da requisição que chega no método `authorize`. Ele conterá todas as informações necessárias para o tratamento e geração do token.

Em seguida, verifique se o usuário, cujo e-mail foi fornecido nas credenciais, realmente está cadastrado no Datastore:

```
Entity userEntity = findUserByEmail(oauthRequest.getUsername());
if (userEntity == null) {
    return buildInvalidUserPassResponse();
}
```

Depois, verifique se o tipo da credencial fornecida está correto, assim como se senha do usuário bate com o que está cadastrado no Datastore:

```
if (oauthRequest.getParam(OAuth.OAUTH_GRANT_TYPE).equals(
        GrantType.AUTHORIZATION_CODE.toString())) {
    if (!checkAuthCode(oauthRequest.getParam(OAuth.OAUTH_CODE))) {
        return buildBadAuthCodeResponse();
    }
} else if (oauthRequest.getParam(OAuth.OAUTH_GRANT_TYPE).equals(
        GrantType.PASSWORD.toString())) {
    if (!userEntity.getProperty(UserManager.PROP_PASSWORD).equals(
oauthRequest.getPassword())) {
        return buildInvalidUserPassResponse();
    }
}
```

Se tudo estiver correto, gere o token e responda de volta, informando o valor da chave de acesso, o tipo do token e o tempo em que ele será válido (nesse caso, 3600 segundos):

```
final String accessToken = oauthIssuerImpl.accessToken();

if (saveToken(oauthRequest.getUsername(), accessToken)) {
    OAuthResponse response = OAuthASResponse
            .tokenResponse(HttpServletResponse.SC_OK)
            .setAccessToken(accessToken)
            .setExpiresIn("3600")
            .setTokenType("bearer")

            .buildJSONMessage();
    return Response.status(response.getResponseStatus())
            .entity(response.getBody()).build();
} else {
    return buildInternalServerError();
}
```

Repare no teste e na chamada ao método `saveToken`. Ele é usado para salvar o token criado no MemCache, associando com o endereço de e-mail do usuário. Na próxima seção, isso será explicado melhor.

Caso alguma exceção aconteça, como má formação da credencial ou informação faltando, responda com uma mensagem de erro apropriada:

```
} catch (OAuthProblemException e) {
    OAuthResponse res = OAuthASResponse
            .errorResponse(HttpServletResponse.SC_BAD_REQUEST).err
or(e)
            .buildJSONMessage();
    return Response.status(res.getResponseStatus())
            .entity(res.getBody()).build();
}
```

Os métodos privados `buildBadAuthCodeResponse()`, `buildInvalidUserPassResponse()` e `buildInternalServerError()` foram utilizados nos trechos de código mostrados para geração de mensagens de resposta de erro,

caso algo desse errado, como senha inválida ou erro de parser da credencial de acesso.

Também foram mostrados os métodos privados `checkAuthCode` para verificar o tipo de token de autenticação (Bearer token), e `findUserByEmail()` para localizar o usuário no Datastore através de seu e-mail.

Esses códigos, assim como o restante do projeto, estão no repositório de código do livro, no endereço: https://github.com/siecola/GAEBook.

Armazenando tokens no MemCache

Quando o token é gerado no método `authorize`, outro método é chamado para salvá-lo no serviço MemCache do Google App Engine. O MemCache foi escolhido para guardar o token gerado por duas razões: para que não haja acessos de escrita e leitura no Datastore sempre que o token for gerado e verificado a cada requisição, e para utilizar seu mecanismo de expiração configurável, fazendo que o token possa deixar de existir após esse tempo, sem que você tenha de escrever códigos para lidar com isso.

Para facilitar o trabalho com o MemCache, crie um `Singleton` para fornecer a instância do MemCache, já configurado com o tempo de expiração dos valores que nele serão inseridos. Para isso, cria a classe `TokenCacheManager` no pacote `com.siecola.exemplooauth`, como mostra o trecho a seguir:

```
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.TimeUnit;

import javax.cache.Cache;
import javax.cache.CacheException;
import javax.cache.CacheFactory;
import javax.cache.CacheManager;
```

```
import com.google.appengine.api.memcache.jsr107cache.GCacheFactory
;

public class TokenCacheManager {

    private static TokenCacheManager instance = null;

    private static Cache cache;

    private TokenCacheManager(){}

}
```

Para pegar a instância de `TokenCacheManager` e do objeto `Cache` para acessar o MemCache, crie o método `getInstance`, como mostrado a seguir:

```
public static TokenCacheManager getInstance() throws CacheExceptio
n {
    if (instance == null) {
        CacheFactory cacheFactory = CacheManager.getInstance().get
CacheFactory();
        Map properties = new HashMap<>();
        properties.put(GCacheFactory.EXPIRATION_DELTA, TimeUnit.HO
URS.toSeconds(1));
        cache = cacheFactory.createCache(properties);
        instance = new TokenCacheManager();
    }

    return instance;
}
```

A configuração do tempo de expiração dos valores a serem inseridos no MemCache é feita pela criação da propriedade `GCacheFactory.EXPIRATION_DELTA`, que expressa o tempo em segundos. Como você pode ver, agora a instância do MemCache que será usada na aplicação `exemplooauth` está configurada para expirar os valores nele inseridos em uma hora, ou seja, 3600 segundos, como foi configurado na resposta do método `authorize` da classe `OAuthToken`.

Por fim, para ter acesso à instância de `Cache` que foi criada no

método `getInstance` , crie o método `getCache` para retorná-la:

```
public Cache getCache() {
    return this.cache;
}
```

Agora toda a aplicação poderá utilizar o `Singleton` `TokenCacheManager` , que traz uma instância de `Cache` já configurada.

13.5 CRIANDO A CLASSE DE FILTRO

A classe de filtro do projeto `exemplooauth` será bem parecida com a que foi criada no projeto `exemplo1` , com apenas algumas alterações, pois agora ela retirar um token da requisição, e não mais o e-mail e a senha do usuário.

Para isso, crie o pacote `com.siecola.exemplooauth.authentication` e a classe `AuthFilter` , da mesma forma como foi feito no projeto `exemplo1` :

```
public class AuthFilter implements ContainerRequestFilter {

    private static final String ACCESS_UNAUTHORIZED = "Você não te
m permissão para acessar esse recurso";

    @Context
    private ResourceInfo resourceInfo;

    @Override
    public void filter(ContainerRequestContext requestContext) {

    }
}
```

O início desse método é exatamente igual ao do projeto `exemplo1` , onde a presença da anotação `@PermitAll` é verificada, assim como a presença do cabeçalho `Authorization` , que carregará o token obtido pelo usuário:

```
Method method = resourceInfo.getResourceMethod();

if (method.isAnnotationPresent(PermitAll.class)) {
    return;
}

String auth = requestContext.getHeaderString("Authorization");

if (auth == null) {
    requestContext.abortWith(Response.status(Response.Status.UNAUT
HORIZED)
            .entity(ACCESS_UNAUTHORIZED).build());
    return;
}
```

Em seguida, pegue o token do cabeçalho `Authorization` e passe para o método `checkCredentialsAndRoles` , que deverá ser alterado para receber o valor do token em vez do e-mail e a senha do usuário:

```
String accessToken = auth.substring("Bearer ".length());

RolesAllowed rolesAllowed = method.getAnnotation(RolesAllowed.clas
s);
Set<String> rolesSet = new HashSet<String>(Arrays.asList(rolesAllo
wed.value()));

if (checkCredentialsAndRoles (accessToken, rolesSet, requestContex
t) == false) {
    requestContext.abortWith(Response.status(Response.Status.UNAUT
HORIZED)
            .entity(ACCESS_UNAUTHORIZED).build());
    return;
}
```

Validando os tokens no MemCache

Agora o método `checkCredentialsAndRoles` deverá receber o valor do token enviado na requisição, procurando-o no MemCache, caso ainda não tenha expirado. Após isso, é necessário localizar o usuário por meio do seu endereço de e-mail, que está atrelado ao token no MemCache:

```
private boolean checkCredentialsAndRoles (String accessToken, Set<
```

```
String> roles, ContainerRequestContext requestContext) {
    boolean isUserAllowed = false;
    String email;
    try {
        Cache cache = TokenCacheManager.getInstance().getCache();

        if (cache.containsKey(accessToken)) {
            email = (String) cache.get(accessToken);
            DatastoreService datastore = DatastoreServiceFactory
                    .getDatastoreService();
            Filter emailFilter = new FilterPredicate(UserManager.P
ROP_EMAIL, FilterOperator.EQUAL, email);
            Query query = new Query(UserManager.USER_KIND).setFilt
er(emailFilter);
            Entity userEntity = datastore.prepare(query).asSingleE
ntity();

            if (userEntity != null) {
                if (roles.contains(userEntity.getProperty(UserMana
ger.PROP_ROLE))) {
                    final User user = updateUserLogin(datastore, u
serEntity);
                    requestContext.setSecurityContext(new Security
Context() {
                        @Override
                        public boolean isUserInRole(String role) {
                            return role.equals(user.getRole());
                        }
                        @Override
                        public boolean isSecure() {
                            return true;
                        }
                        @Override
                        public Principal getUserPrincipal() {
                            return user;
                        }
                        @Override
                        public String getAuthenticationScheme() {
                            return SecurityContext.BASIC_AUTH;
                        }
                    });
                    isUserAllowed = true;
                }
            }
        }
    } catch (CacheException e) {
    }
    return isUserAllowed;
```

```
}
```

Depois que o usuário é localizado no Datastore pelo seu e-mail que estava atrelado ao token no MemCache, o código segue sua execução da mesma forma como foi feito no projeto `exemplo1`, inclusive chamando o método `updateUserLogin` para implementar o mecanismo da lógica de salvar a data/hora do último login do usuário.

Tudo pronto! Agora é só colocar a aplicação `exemplooauth` em execução e testar o mecanismo de autenticação OAuth 2 que foi implementado nela.

13.6 TESTANDO COM O REST CONSOLE

É possível testar o mecanismo de autenticação OAuth 2 da aplicação, utilizando o REST Console. Para isso, execute-a no Eclipse na sua máquina de desenvolvimento.

> Se a aplicação `exemplooauth` estiver no mesmo workspace do Eclipse do projeto `exemplo1`, remova-o do servidor na aba `Servers` e adicione o projeto `exemplooauth`.

Para adquirir o token de acesso, é necessário acessar o serviço que foi criado como parte do provedor de autenticação. Para isso, configure o REST Console para realizar um `POST` no endereço http://localhost:8888/api/token.

Requisitando o token com o REST Console

Ele deverá ser configurado para aceitar uma resposta com o conteúdo no formato `application/json` e enviar no corpo da

requisição as credenciais no formato `application/x-www-form-urlencoded` . Tais credenciais, como já foi explicado, deverá ter os seguinte formato:

```
grant_type=password&username=admin@siecola.com&password=Admin#7&cl
ient_secret=Admin#7&client_id=admin@siecola.com
```

Veja como fica configurado o REST Console na seção `Target` para satisfazer esses requisitos:

Figura 13.1: Solicitando token com REST Console - Target.

E na seção `Body` , onde as credenciais para solicitar o token deverão ficar:

Figura 13.2: Solicitando token com REST Console - Body.

A resposta à requisição ao serviço que fornece o token é no

formato JSON, como mostrado a seguir:

```
{
    "token_type": "bearer",
    "expires_in": 3600,
    "access_token": "3ec10cd3f53a311e87dd4097f2ed2fb5"
}
```

Nele, há o tipo do token obtido, que é o Bearer, o tempo em segundos que esse token é válido no provedor de autenticação e o valor da chave de acesso em si.

Figura 13.3: Token gerado

Acessando o serviço de usuários com o token obtido

Agora que o token já foi obtido no provedor de autenticação, é possível utilizá-lo para acessar o serviço de usuários ou qualquer outro que você crie na aplicação `exemplooauth`, que esteja no pacote `com.siecola.exemplooauth.services`. Para isso, basta incluir, em todas as requisições, o cabeçalho `Authorization` no seguinte formato:

```
Authorization: Bearer token_de_acesso
```

Onde o `token_de_acesso` é o valor da chave obtida do provedor de autenticação. Veja como fica o REST Console, configurado para tal:

Figura 13.4: Campo Authorization

13.7 COMPORTAMENTO DA APLICAÇÃO CLIENTE

Os passos que foram descritos na seção anterior são exatamente aqueles que uma aplicação cliente, desenvolvida em Java, Android, Swift ou qualquer outra linguagem, deve executar para poder adquirir o token de acesso e usá-lo em todas as requisições aos serviços providos por uma aplicação que implementa o mecanismo OAuth 2 com Bearer token.

Além disso, a aplicação cliente também deve se atentar ao fato que o token vai expirar no provedor de autenticação. Por isso, quando uma tentativa de acesso a alguma serviço falhar e ela receber um o código `HTTP 401 UNAUTHORIZED` , um novo token deve ser requisitado.

13.8 CONCLUSÃO

E é isso! Agora você pode construir serviços REST, hospedados no Google App Engine, de forma mais segura com o mecanismo de

autenticação OAuth 2 com Bearer token. Um mecanismo muito utilizado e conhecido.

Capítulo 14

ALGO MAIS SOBRE GOOGLE APP ENGINE

O Google App Engine é uma plataforma fantástica que sempre está inovando e lançando novas funcionalidades. Há algumas outras coisas que você pode se interessar ou talvez necessite para um outro projeto. Veja a seguir alguns temas e uma referência de consulta na documentação do Google.

Envio e recebimento de e-mails

É possível enviar e receber e-mails de uma aplicação do Google App Engine. Para maiores informações, consulte https://cloud.google.com/appengine/docs/java/mail/.

Google Cloud SQL

O Google oferece uma infraestrutura de hospedagem de banco de dados, semelhante ao MySQL. Com ele, você pode desenvolver aplicações que necessitem de uma base de dados relacional ou que um dia poderão ser portadas para outra plataforma. Para maiores informações, consulte https://cloud.google.com/appengine/docs/java/cloud-sql/.

Google Cloud Endpoints

Esse é um framework para desenvolvimento de APIs na parte do

back-end de uma aplicação do GAE, para ser utilizado por clientes móveis como Android e iOS. O conceito é interessante e agiliza muito o desenvolvimento de ambas as partes, mas é necessário avaliar com cuidado sua utilização se você deseja desenvolver uma aplicação que possa ser usada por uma gama maior de clientes desenvolvidos em outras tecnologias e linguagens.

Para maiores informações, consulte https://cloud.google.com/appengine/docs/java/endpoints/.

JPA e JDO

É possível utilizar JPA ou JDO para acessar os dados no Google Cloud Datasotre, abstraindo a API de baixo nível nativa do SDK.

Para informações sobre como utilizar JDO, consulte https://cloud.google.com/appengine/docs/java/datastore/jdo/overview.

Para informações sobre como utilizar JPA, consulte https://cloud.google.com/appengine/docs/java/datastore/jpa/overview-dn2

14.1 CONCLUSÃO

Lembre-se de que você pode encontrar tudo o que foi desenvolvido aqui no repositório de código do livro no endereço: https://github.com/siecola/GAEBook.

Você também pode participar do Fórum da Casa do Código, deixando comentários, dúvidas ou sugestões. O link é: http://forum.casadocodigo.com.br.

Espero que vocês tenham gostado e aproveitado do livro! Até o próximo!